UNIVERSITY
of
GLASGOW

Computing Science

Ph.D Thesis

Sort Inference in Action Semantics

Deryck Forsyth Brown

Submitted for the degree of

Doctor of Philosophy

Thesis
10804
Copy 1

Sort Inference in Action Semantics

by
Deryck Forsyth Brown
Submitted to the Department of Computing Science
on 30th September, 1996
for the degree of
Doctor of Philosophy

Abstract

Action semantics is a semantic meta-language developed by Mosses and Watt for specifying programming languages. The work reported in this thesis is part of a project to develop a system, called ACTRESS, that is a semantics-directed compiler generator based on action semantics. The aims of this project are to demonstrate the suitability of action semantics for this task, and to produce a system that improves on the performance of previous semantics-directed compiler generators. Moreover the ACTRESS system aims to accept a wide range of programming languages, including dynamically-scoped and dynamically-typed languages, but not to penalise the implementations of statically-typed or statically-bound languages as a result.

ACTRESS automatically generates a compiler from an action semantic description of a programming language, and has been used to generate compilers for a small declarative language and a small imperative language. The generated compiler uses a number of standard modules to compile the action denoting a program into efficient object code. Amongst these modules is the action notation sort checker. The role of the action notation sort checker is vital. It analyses an action and infers detailed information about the sorts of data flowing between the sub-actions. Without this information, erroneous actions could not be detected, and efficient code generation would not be possible.

The problem of sort inference for action notation is a complicated one. Firstly, action notation has an unusual sort system, which includes individuals as sorts, sort join, and sort meet. Secondly, the complex data flows in action notation prevent a simple bottom-up or top-down analysis. In general, actions have polymorphic sorts. Thirdly, we aim to be as general as possible, and allow actions that still require sort checks when the action is performed. We must detect the places in an action where a run-time sort check is necessary, and annotate the action accordingly.

In this thesis, we present a sort inference algorithm for action notation, that is specified as a collection of sort inference rules, and we describe the implementation of this algorithm to produce the action notation sort checker. Furthermore, we formulate a soundness property for our sort inference algorithm, and prove its soundness with respect to the natural semantics of ACTRESS action notation.

Finally, we compare ACTRESS with other semantics-directed compiler generators that use action semantics, and suggest possible improvements and future research.

Thesis Supervisor: Prof. David A. Watt

To my parents,
James and Jean.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, David Watt, for his encouragement and guidance throughout the course of my studies. In particular, if it was not for his advice, patience and dedication as a supervisor, then I doubt this thesis would ever have been produced. David, you have my eternal gratitude.

I would also like to thank David Schmidt for his assistance in formulating the soundness property, and advice in constructing the proof. I gladly acknowledge his contribution to this work.

I would also like to thank the following people for their interesting conversations and helpful comments on my work: Olivier Danvy, Kyung-Goo Doh, Jan-Jaap IJdens, John McCall, Peter Mosses, Hermano Moura, David Schmidt, Carron Shankland, Phil Wadler, and Phil Watson.

During my studies, I have received financial assistance from a number of sources: SERC provided the financial support for my studies; SERC, Glasgow University, and Kansas State University paid for my visit to Kansas; and Glasgow University and The Robert Gordon University paid for my stay during the Glasgow University Computing Science Department Research Festival. For all of their contributions, I am extremely grateful.

I would also like to thank my friends and colleagues at The Robert Gordon University. For encouraging me during the writing process, I would especially like to thank: David Harper, Susan Craw, Dean Henderson, David Hendry, Jan-Jaap IJdens, and Arturo Trujillo. Also, a special thankyou to John McCall for his willingness to listen to my ramblings and act as my consultant mathematician.

To all of my friends at Glasgow University, thankyou for making it such a great place to be. To Trish Cashen, Gilbert Cockton, Phil Gray, Derick Kirkwood, Brian Matthews, Carron Shankland, Duncan Sinclair, and Phil Watson, thanks for providing the lighter side to my studies.

Finally, I would like to thank my mother. Her years of support have made all of my studies possible.

Aberdeen
September, 1996

D.F.B.

Chapter 1

Introduction

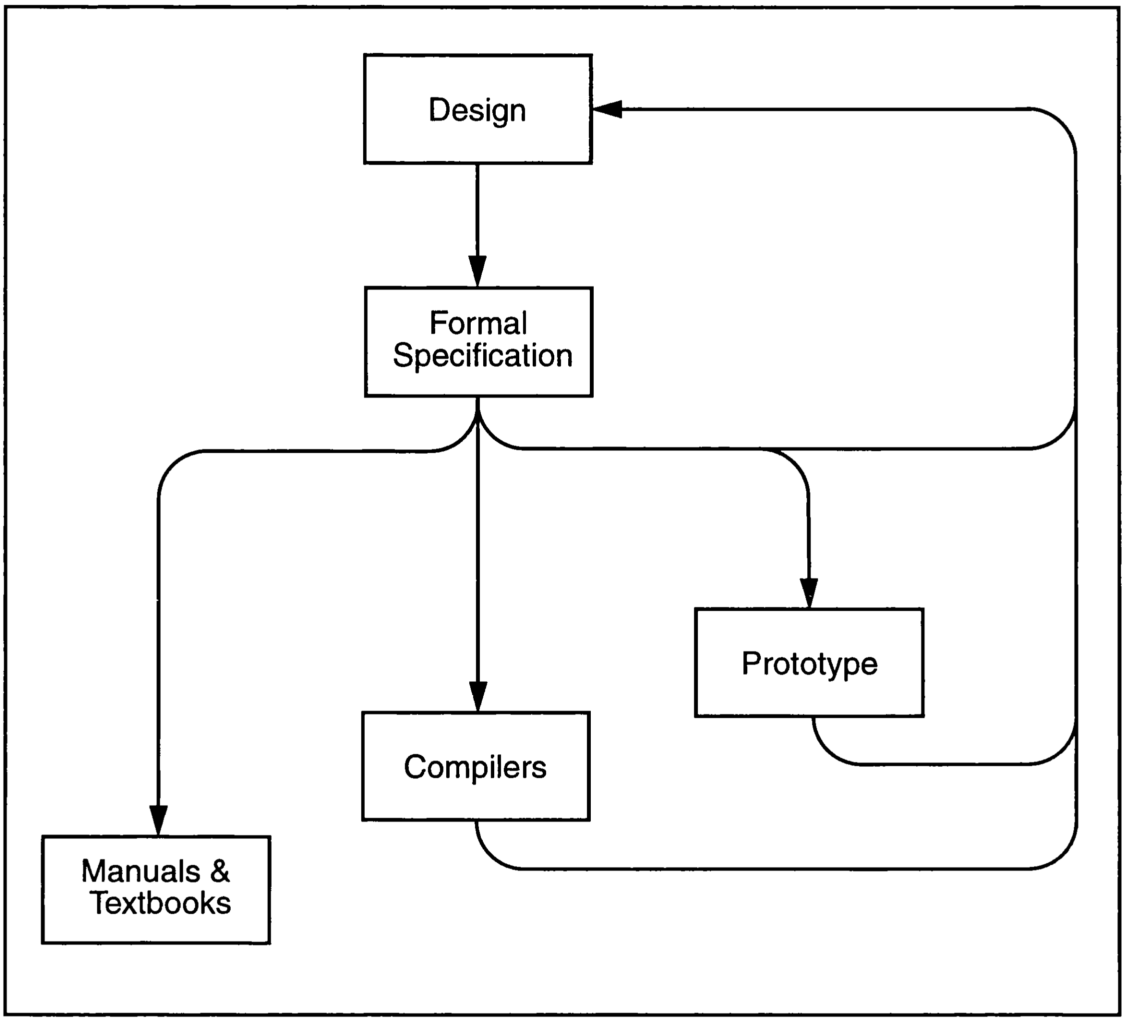
1.1 Programming Language Life-Cycle

Programming language design is one of the hardest skills in computing science. No-one fully understands how certain design choices influence the way in which the programming language is used and the level to which it is adopted.

In an ideal situation, programming language development would follow a life-cycle, just as software development does. An example of such a *programming language life-cycle* is proposed by Watt[Wat90,Wat93,Wat96] and given in Figure 1.1. The main aim is that the language design is *immediately* captured in a formal specification.

The formal specification of the language is useful, as it tends to identify poor design choices. Experience has shown[Wat96] that poor designs are often difficult to specify formally. Of course, the formal specification is also a complete and unambiguous description of the programming language. Ideally this specification should be accessible to a wide readership, so that the design is scrutinized by as many people as possible. This specification should then be used as the basis of a prototype implementation. The prototype implementation serves two purposes: (i) it tests the formal specification, and (ii) it allows programmers to gain initial experience with the new language.

Figure 1.1: The programming language life-cycle



Both the formal specification and the prototype implementation provide timely feedback for the language designer, allowing the language design to be altered, and hopefully improved. The process of design, specification and prototyping can be repeated as often as necessary until the language has stabilised. Only when the language is stable should the much greater task of producing the first production-quality compilers be attempted.

Developing the first production-quality compilers is a much longer process, perhaps taking several years. It is still not too late, however, at this stage to provide feedback to the language design. Of course, it is hoped that any changes made to the language design at this stage are relatively minor ones.

Finally, and possibly in parallel with the first implementations, comes the production of language manuals and textbooks. It is even more important that the language design is fixed at this stage, since changes made after this may not be reflected in textbooks for several years. This is also the best time to produce the language standards document—ideally of course, this is just the same formal specification that has been used throughout the development process.

1.2 Programming Language Specification

Unfortunately, most programming language designers tend not to write any kind of formal specification of the language, either as it is being designed or even retrospectively. This is often understandable, since many formalisms offer poor support for the design process of repeatedly extending and modifying the language. In fact some formalisms often require a complete rewrite of the specification for what appears to be a minor change in the language. For example, classical denotational semantics requires a complete re-write of the specification if a language change forces a move from direct to continuation-passing style.

To be of any practical value, the semantic meta-language should have excellent pragmatic qualities, namely:

Modularity: A specification should be broken down into a collection of independent modules. Just as in software engineering, the interfaces between these modules are carefully designed, and several people can be working on specifying different modules concurrently.

Resilience: The specification should be resilient, i.e. small changes to the language should only involve small changes to the specification. Global changes should rarely (or never) be necessary.

Extensibility: A specification should be easily extended from a simple language to a more complicated one. The language designer should be able to start small and gradually increase the language to its full size.

Re-usability: A specification for a similar language should be re-usable without major changes. Few programming languages are truly original, most borrow heavily from existing languages.

Familiarity: The meta-language should make use of the common concepts in programming languages. Familiar concepts will make the specification more accessible to a wide audience.

Classical denotational semantics possesses none of these qualities: it encourages monolithic specifications; and the underlying semantic domains are visible throughout the specification, either as arguments of semantic functions or through their associated auxiliary operations. Using software engineering terminology, the semantic functions are tightly-coupled to the semantic domains and their operations.

Action Semantics[Wat90,Mos92] is a relatively new semantic meta-language. However, it was designed to possess all of the qualities given above. This makes it an ideal choice for producing the formal specification needed as part of the programming language life-cycle.

1.3 Programming Language Implementation

As we have seen from the programming language life-cycle, a *rapid* and *correct* prototype implementation is necessary at an early stage in the design process. Moreover, this prototype implementation must be able to keep up with changes in the language design. Finally, the prototype implementation must be *efficient* to allow useful experiments to be conducted with the language. Ideally, to provide such an implementation, the prototype should be generated automatically using tools, rather than written by hand. However, the type of compiler generation system used is also important.

Initial work in compiler generation was directed towards *compiler writing systems.* A compiler writing system is a collection of one or more tools that can be used to generate the parts of a compiler. These systems are essentially customized languages for expressing compilers in a sufficiently abstract and flexible way, including very low-level information about such things as syntax trees, instruction selection, and register allocation. Compiler writing systems are extremely important when it comes to producing the first production-quality compilers for the programming language. Such systems can substantially reduce the time taken to develop a compiler. Unfortunately, they are of little use when constructing the prototype implementation—they do not provide a means of automatically and rapidly generating a compiler from the language's specification. These are, however, the aims of *semantics-directed compiler generation.*

Semantics-directed compiler generation is not a new topic in computing science. The goal of automatically generating a compiler from a specification of a programming language has been pursued ever since semantic meta-languages were developed. Unlike a compiler writing system, however, a semantics-directed compiler generation system takes as input a *formal specification* of the programming language and generates a compiler from it. Such a generated compiler must be good enough to use as a prototype implementation. For example, it should not be more than an order of magnitude slower than a hand-written compiler, otherwise it will be difficult to perform useful experiments with the language. Ideally, such a generated compiler should also be efficient enough to compete with either a hand-written compiler, or one generated using a compiler writing system. This would allow for the extremely fast development of *correct* compilers—something which the other approaches lack.

Unfortunately, in semantics-directed compiler generation, progress to date has been disappointing. Typical systems generate a compiler that produces code that is two or three orders of magnitude slower than a hand-written compiler. Compile times are similarly disappointing. Moreover, the effectiveness of a compiler generation system seems to depend heavily on the type of formal specification it processes. We aim to improve this by using action semantics as the basis for our compiler generation system.

In this thesis, we consider a new semantics-directed compiler generator called ACTRESS. ACTRESS processes an action-semantic specification of a programming language, and automatically generates a compiler for that language, expressed in STANDARD ML. More precisely, the thesis is concerned with the problem of sort inference in action notation. In the ACTRESS system, sort inference performs a role that is analogous to type checking in an ordinary compiler for a programming language. Also, just as in an ordinary compiler, the sort information collected during sort inference is of vital importance during the later stages of the ACTRESS system.

1.4 Outline of Thesis

1.4.1 Readership

This thesis deals with the problem of sort (or type) inference in action notation. As such, we assume that the reader is familiar with the related problem of type inference in declarative programming languages. This work also makes extensive use of the record types introduced by Wand[Wan87,Wan89] (and others), and some prior knowledge would be beneficial. Finally, this thesis cannot give more than a brief introduction to the subject of action semantics. For more information, the reader is directed to [Wat90,Mos92].

1.4.2 Organization

The remainder of this thesis is structured as follows:

- Chapter 2 gives a brief introduction to action semantics in general, and the subset of action semantics used in the ACTRESS system in particular. It also introduces the formal semantics of the ACTRESS subset of action notation.

- Chapter 3 details the previous work in semantics-directed compiler generation including compiler generation systems based on denotational semantics, partial evaluation, high-level semantics, and action semantics.
- Chapter 4 describes the ACTRESS system. It presents the overall structure of ACTRESS as a collection of modules, and then discusses the role of each module in detail.
- Chapter 5 introduces the notion of sort in action notation and in the ACTRESS system. It uses record sort schemes to model the data flows between actions.
- Chapter 6 describes our sort inference algorithm for the ACTRESS subset of action notation. This algorithm is formalised as a set of sort inference rules, expressed in terms of record sort schemes.
- Chapter 7 proves the soundness of the sort inference algorithm with respect to the formal semantics of the ACTRESS subset of action notation.
- Chapter 8 concludes and gives suggestions for further research.
- Appendix A gives the formal semantics of the ACTRESS subset of action notation. It is presented as a natural semantics using inference rules.
- Appendix B gives the sort inference rules used to specify the sort inference algorithm.
- Appendix C gives the syntactic specification for ACTRESS action semantic specifications.
- Appendix D gives the proof of commutativity for the *meet* operation, and the

proof of normalisation for the *normalise* operation given in Chapter 5.

Chapter 2

Action Semantics

2.1 Introduction

Action semantics is concerned with giving a formal meaning to the programs of a programming language. Just as a program is constructed from separate phrases: statements, expressions, procedures, etc., so the meaning of a complete program, or its *denotation*, can be constructed from the meanings of its separate phrases.

The specification of a programming language can be decomposed into three different components:

- *syntax*: the specification of a grammar that defines the legal programs of the language.

- *static semantics*: the specification of the checks (or constraints) that a program must satisfy before it can be run.

- *dynamic semantics*: the specification of the meaning of executing a program. Only programs that satisfy the static semantics need to be given a meaning. This can be further split into two sub-parts:

  - semantic entities: the specification of the data objects and associated operations that the programming language manipulates.
  - semantic functions: a collection of mutually-recursive functions that map

programs to their denotations. The use of semantic functions is characteristic of *a denotational semantics.*

In action semantics, these three components are all specified within an algebraic framework called *unified algebras*. For simplicity, however, we only concern ourselves with the syntax and dynamic semantics of a programming language. Since an action semantics uses semantic functions, it is a denotational semantics.

Action semantics is constructed as two standard specifications within the framework of unified algebras. The first, called *data notation*, specifies a collection of common sorts and operations on them. The second, called *action notation*, specifies a collection of primitive actions and action combinators that can be used to construct the denotation of a program.

This chapter can only be a very brief introduction to the area of action semantics, and the details of data notation and action notation. For a full description, the reader is directed to [Mos92,Wat91].

In the following sections we introduce data notation and action notation in turn. Finally, we show how action semantics can be used to specify the syntax and dynamic semantics of an example language, NANO–Δ.

2.2 Data Notation

In action semantics, we have a collection of pre-defined sorts of data along with operations over them. Collectively, these form the *data notation* of action semantics. We can also define our own sorts and operations using the notation of unified algebras.

The standard data notation is specified in Appendix B of [Mos92]. It includes specifications for truth-values, natural numbers, integers, characters, lists, strings, tuples, sets, maps and trees. Some examples of standard data notation are:

- 1, true, "x": these are just simple values.
- integer, truth-value, list[integer]: these are sorts of values. The sort "list[integer]" is the sort of all lists with integer elements.
- sum(1, 3), either(true, false), list of ("a", "b", "c"): these are data operations. The operation "list of ("a", "b", "c")" produces the 3-element list "["a", "b", "c"]".

The specification of a programming language requires the definition of *semantic entities*. These are the particular sorts of information that the programming language manipulates. They typically include primitive sorts such as integers and truth-values, as well as more complicated sorts such as records and arrays. The standard data notation allows us to re-use the specifications for primitive values it provides, and to construct new data sorts using the existing definitions as a basis.

ACTRESS data notation is a restricted form of the standard data notation, and includes integers, truth-values, and lists. The subset of the standard data notation allowed in ACTRESS was carefully chosen to simplify the implementation of the data operations, whilst still allowing useful language specifications to be written.We discuss some limitations on the names of data operations in ACTRESS data notation in Chapter 4, and we discuss the sorts of data in ACTRESS data notation in Chapter 5.

2.3 Action Notation

We begin by giving an overview of the features of standard action notation. In Sections 2.3.2 to 2.3.8, we introduce ACTRESS action notation, the subset of action notation used in the ACTRESS system.

2.3.1 Standard Action Notation

In action semantics, the denotation of a program is an *action*. An action is a computational entity that can be *performed*, given some *incomes*, to produce an *outcome*. An action is formed by composing primitive actions using action combinators.

The performance of an action can have a number of different outcomes:

- it can *complete*, i.e. terminate normally;
- it can *escape*, i.e. terminate exceptionally;
- it can *fail*, i.e. terminate erroneously;
- it can *diverge*, i.e. not terminate.

In action notation, both escape and failure can be detected and the performance of an enclosing action continued. In the case of escape, the enclosing action resumes from the point at which the escape is explicitly *trapped*. In the case of failure, the enclosing action will continue with the performance of an alternative (if one exists). Alternatives arise through the use of certain action combinators.

Actions operate on a number of different kinds of information:

- *transient* information is a tuple of data, i.e. intermediate results.
- *scoped* information is a set of bindings from tokens to data values, i.e. a symbol table or environment.
- *stable* information is stored in cells, i.e. values assigned to variables.
- *permanent* information represents messages passed between several actions.

An action is committed to the current alternative once it has made a change in either stable or permanent information. Such a change cannot be reversed, and so the action cannot back-track and try another alternative in the event of failure.

These different kinds of information give rise to the different *facets* of an action:

- the *basic* facet deals with the control flow within the action.
- the *functional* facet deals with transient information: actions give and are given transients.
- the *declarative* facet deals with scoped information: actions produce and receive bindings.
- the *imperative* facet deals with stable information: actions reserve and unreserve storage, and change the data stored in cells.
- the *communicative* facet deals with permanent information: actions send and receive messages, and offer contracts to agents.

Some actions contain terms whose value depends on the information supplied to the action when it is performed. Examples include accessing a particular binding, or inspecting the current contents of a storage cell. Such terms are called *yielders*. Action notation has primitive yielders for each of its facets. Moreover, a data operation becomes a yielder whenever any of its arguments are yielders.

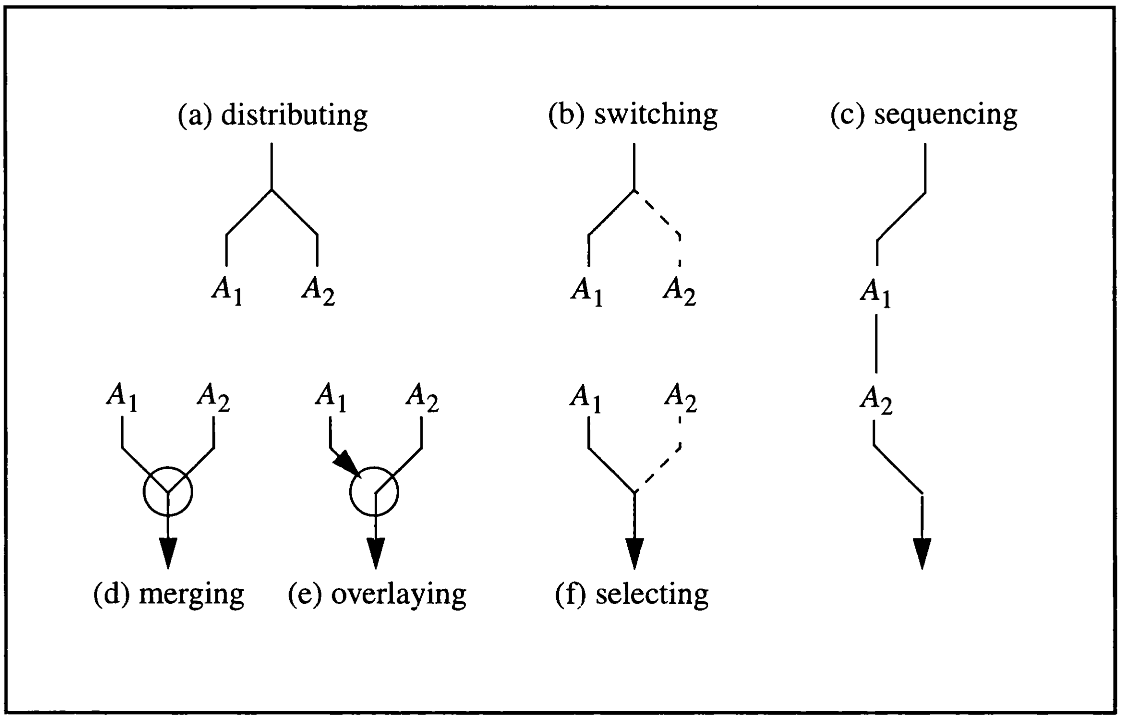
There are a number of possible data flows in action notation. These data flows determine how the information received by a compound action is propagated to its sub-actions, and how the results of performing the sub-actions are combined to produce the information generated by the compound action. The possible data flows found in action notation, illustrated in Figure 2.1, are as follows:

(a) *Distributing*: the input information is propagated to both A_1 and A_2.

(b) *Switching*: the input information is propagated to either A_1 or A_2 (where only one of these actions will be performed).

(c) *Sequencing*: the input information is propagated to A_1 only, the output information from A_1 is propagated as the input information to A_2, finally, the

Figure 2.1: Data flows in action notation



output information from A_2 is the overall output.

(d) *Merging*: the output information from A_1 and A_2 is combined, and the domains of the outputs must be disjoint.

(e) *Overlaying*: the output information from A_1 and A_2 is combined, and the output information from A_2 takes precedence over that from A_1.

(f) *Selecting*: the output information is chosen from that produced by either A_1 or A_2 (where only one action has been performed).

Each action combinator will use one of these possible data flows for transients and bindings (the different combinations yield different action combinators). Most action combinators use sequencing for storage.

Actions can be classified by their *incomes*, the kinds of information they receive, and their *outcomes*, the kinds of information they produce on termination. For example, the sort of an action denoting the evaluation of an expression might be:

action [giving a value] [using current bindings|current storage]

This describes an action which receives both bindings and storage, and which (on completion) gives a single transient of sort value. Also, the sort of an action denoting the execution of a statement might be:

action [storing|diverging] [using current bindings|current storage]

Here, the action has the same incomes as before, but its outcomes allow it to make changes to storage. Non-termination is also given as a possible outcome, since a while-statement may loop indefinitely. The absence of functional and declarative outcomes indicates that it gives no transients or bindings on completion. Also for expressions, the absence of storing as a possible outcome implies expression evaluation in this example is free from side-effects. Finally, the sort of an action denoting the elaboration of bindings might be:

action [storing|binding] [using current bindings|current storage]

Here again, the action has the same incomes as the previous ones. However, its outcomes allow it to both modify storage and produce bindings. The need to modify storage arises from the allocation of new storage cells for the variables being declared, rather than from modifying the values of existing variables. Note that failure is an implicit outcome in all actions, so a program could still terminate erroneously. We consider the sorts of actions in Chapter 5.

2.3.2 ACTRESS Action Notation

ACTRESS action notation, which is described in the following sections, is derived from an earlier version[1] of the notation that is presented in [Mos92]. Its main difference lies in the treatment of transients. In ACTRESS action notation, transients are treated more like bindings, i.e. they are represented as a set of values labelled by natural numbers, rather than tuples. The other differences are purely syntactic.

[1]. Specifically, draft version 8 (Autumn 1990) of Mosses' book.

In ACTRESS, we do not consider actions that can escape, nor do we consider the communicative facet. Moreover, we do not distinguish between committed and uncommitted failures when an action is being performed. All run-time failures are treated as committed failure. This allows us to ignore the potential for back-tracking within the performance of an action. These restrictions, however, still allow useful language specifications to be written.

2.3.3 Basic Action Notation

Basic action notation is concerned with the temporal ordering of the performance of the sub-actions in an action. The basic actions and combinators are therefore most easily demonstrated by the programming language constructs that deal with control flow.

The simplest primitive action is "complete". It simply terminates normally, giving no transients, producing no bindings, and making no changes to storage. It is used to specify the semantics of a NANO-Δ skip-statement:

- execute [["skip"]] = complete .

The action "A_1 and then A_2" causes the action A_1 to be performed before the action A_2. The transients and bindings are distributed to the actions A_1 and A_2. The transients and bindings produced by A_1 and A_2 are merged. The store is sequenced from A_1 to A_2.

The "and then" combinator is used to specify the semantics of a NANO-Δ statement sequence:

- execute [[S_1:Statement ";" S_2:Statement]] =
  execute S_1
  and then execute S_2 .

In standard action notation, the action "A_1 and A_2" allows the performance of the two sub-actions to be interleaved. The data flows for transients and bindings are the same as "and then", but the order of storage modifications is affected. For example, it is used in the semantics of the NANO-Δ plus-expression where the order of evaluation of the sub-expressions is unimportant. In ACTRESS action notation, we ignore the possibility of interleaved performance, so "and" is treated identically to "and then".

The action "A_1 or A_2" represents the non-deterministic choice between actions A_1 and A_2. One action is chosen, and performed. If it does not fail, then it determines the outcome of "A_1 or A_2". If it does fail, then the other sub-action is performed. If both sub-actions fail, then "A_1 or A_2" fails. The received transients, bindings and store are switched between the actions A_1 and A_2. The transients, bindings and store produced are selected from those produced by A_1 or A_2.

The action "unfolding A" is used to specify iteration. The action A is performed, but whenever the dummy action "unfold" is encountered in A, it is replaced by A, i.e. the action is *unfolded* a further iteration. The (initial) action A receives the same transients, bindings and store as "unfolding A", and "unfolding A" produces the same transients, bindings and store as A.

The "unfolding" combinator is used to specify the semantics of a NANO-Δ while-statement:

```
•  execute [[ "while" E:Expression "do" S:Statement ]] =
        unfolding
        | evaluate E then
        | | execute S and then unfold
        | | else complete .
```

Here, the sub-action of the "unfolding" denotes the body of the loop. The loop begins with an evaluation of the controlling expression E, which should give either true or false. If the result is true, then the first alternative of the "else" is performed, executing the statement S followed by an "unfold" which performs another iteration of the loop. If the result is false, then the action takes the second alternative and completes.

2.3.4 Functional Action Notation

The functional facet is concerned with transient information. Transient information is represented by a map from natural numbers, called labels, to values of sort datum.

The action "give *Y*" evaluates the yielder *Y* to yield a datum, and gives this datum as a single transient labelled 0. If *Y* yields nothing, then the action fails.

The "give" action is used in the evaluation of literals in NANO-Δ:

- evaluate *L*:Literal = give valuation of *L* .

Here, "valuation of" is an auxiliary operation that maps the syntactic form of a literal into its semantic value.

The yielder "the *S*" yields the datum labelled 0 from the current transients, restricted to the sort *S*. Therefore, if the datum is not of sort *S*, then the yielder yields nothing.

The action "A_1 then A_2" sequences the transients between A_1 and A_2. In all other respects, it is the same as "A_1 and then A_2". The "then" combinator is used in the semantics of a negation-expression in NANO-Δ:

- evaluate [["–" *E*:Expression]] =
  evaluate *E*
  then give negation(the integer) .

Here, the single transient given by the evaluation of *E* is passed into the "give" action which gives the negation of the value. If the evaluation of *E* yields a truth-value instead of an integer, then the yielder "the integer" yields nothing and the "give" action fails.

If an action gives more than one transient, then the transients must be explicitly labelled. The action "give *Y* label #*n*" is similar to "give *Y*" except that the resulting transient is labelled by the natural number *n*. The yielder "the *S* #*n*" is similar to the "the *S*", except that it selects the datum labelled *n* from the incoming transients. These points are illustrated by the semantics of the NANO-Δ plus-expression:

- evaluate [[E_1:Expression "+" E_2:Expression]] =

```
    | | evaluate E1 then give the integer label #1
    | and
    | | evaluate E2 then give the integer label #2
    then give the sum of (the integer #1, the integer #2) .
```

Here, both of the expressions E_1 and E_2 are evaluated to give transients labelled 1 and 2 respectively. These are then propagated to the final sub-action which calculates their sum.

The yielder "it" is an abbreviation for "the datum #0".

2.3.5 Declarative Action Notation

The declarative facet is concerned with bindings. Bindings are represented as a mapping from tokens to values of sort bindable.

The action "bind k to Y" evaluates the yielder Y to yield a datum, and then produces a single binding from token k to this datum. This is used in a NANO-Δ constant declaration:

- elaborate [["const" I:Identifier "~" E:Expression]] =
  evaluate E then bind I to the value .

Here, the expression E is evaluated and the datum is passed to the "bind" action. This creates a binding to the token I representing the program identifier.

The action "furthermore A" performs the action A and produces the input bindings overlaid by the bindings produced by A.

The action "A_1 hence A_2" sequences the bindings between A_1 and A_2. In all other respects, it is the same as "A_1 and then A_2".

These two combinators are used to specify the effect of entering a new scope in a programming language. For example, this occurs in NANO-Δ with a let-statement:

- execute [["let" D:Declaration "in" S:Statement]] =

```
    | furthermore elaborate D
    hence
    | execute S .
```

Here, the original bindings are overlaid by those produced by "elaborate D" to create a new scope. The statement S is then executed in this scope.

The action "A_1 moreover A_2" performs both A_1 and A_2. It distributes the bindings to A_1 and A_2. The output bindings are those produced by A_1 overlaid by those produced by A_2. The input transients are distributed, and the output transients are merged.

The action "A_1 before A_2" performs A_1 with the original bindings. It then performs A_2 with the original bindings overlaid by those produced by A_1. The output bindings are those produced by A_1 overlaid by those produced by A_2. In all other respects, it is the same as "A_1 and A_2".

The "before" action is used to specify the semantics of a NANO-Δ declaration sequence:

- elaborate [[D_1:Declaration ";" D_2:Declaration]] =

```
    elaborate D1
    before elaborate D2 .
```

Here, not only will the action as a whole produce the combined bindings of the two declarations D_1 and D_2, but D_2 is also allowed to access the bindings produced by D_1.

2.3.6 Imperative Action Notation

The imperative facet is concerned with storage. Storage is represented as a mapping from cells to values of sort storable. Action notation combinators are carefully selected to guarantee that the store is *single-threaded*, i.e. it can be represented as a single mapping that is never copied or combined with other maps. Imperative action primitives operate *indivisibly* on the store, so one imperative action is never performed at the same instant as another.

The action "store Y_1 in Y_2" evaluates the yielder Y_1 to yield a datum d and evaluates the yielder Y_2 to yield a cell c. It then updates the contents of the cell c to the value d. If either of the yielders yield nothing, then the "store" action fails. Also, if the cell c is not in the domain of the current storage, or if the datum d is not of sort storable, then the "store" action fails.

The "store" action is used to specify the semantics of a NANO-Δ assignment statement:

- execute [[*I*:Identifier ":=" *E*:Expression]] =
  evaluate *E* then store the value in the cell bound to *I* .

Here, the expression E is evaluated and then stored in the cell bound to the variable I.

The yielder "the S stored in Y" evaluates the yielder Y to yield a cell c. It then fetches the current contents of c and restricts it to be of sort S. If either c is not in the domain of the current storage, or the contents of c is not of sort S, then the yielder yields nothing.

The "stored in" yielder is used to partly specify the semantics of a NANO-Δ identifier expression:

- evaluate *I*:Identifier =
  give the value bound to *I*
  or give the value stored in the cell bound to *I* .

Here, the identifier I can be bound to either a datum of sort value (representing a constant identifier) or to a datum of sort cell (representing a variable identifier). Since only one of the yielders "the value bound to I" and "the cell bound to I" will yield a datum for a particular binding (the other will yield nothing, and consequently the enclosing action will fail), then the "or" action can be used to select between the two possible cases. In the latter case, the current value of the variable is fetched from storage using the yielder "the value stored in ...".

The action “deallocate Y” evaluates the yielder Y to yield a cell c. It then removes the cell c from the current storage. If the cell c is not in the domain of the current storage, then the action fails.

2.3.7 Reflective Action Notation

Reflective action notation is concerned with *abstractions*. An abstraction is an element of data that *encapsulates* an action. Abstractions can be given as transients, bound to tokens, and stored in cells, just like other data values.

The data operation “abstraction A” creates an abstraction encapsulating the action A.

The yielder “Y_1 with Y_2” evaluates the yielder Y_1 to yield an abstraction A, and evaluates the yielder Y_2 to yield a single datum d. It produces the modified abstraction in which the encapsulated action will receive a single transient d labelled 0 when it is enacted. If either of Y_1 or Y_2 yields nothing, then the whole yielder yields nothing.

The yielder “closure Y” evaluates the yielder Y to yield an abstraction A. It produces the modified abstraction in which the encapsulated action will receive the bindings current at the point of closure when it is enacted.

The action “enact Y” evaluates the yielder Y to yield an abstraction A. It then performs the encapsulated action of A with the transients and bindings supplied by the “with” and “closure” operations. If the performance of the encapsulated action fails, then the “enact” action also fails.

In the action semantics of NANO-Δ, an abstraction is formed by the procedure declaration, and enacted by the procedure call. Their semantics are:

```
• elaborate [[ "proc" P:Identifier "(" I:Identifier ":" T:Type ")" "~" S:Statement ]] =
      bind P to closure abstraction
       | furthermore bind I to the value
       | hence execute S .
```

- execute [[*P*:Identifier "(" *E*:Expression ")"]] =
  evaluate *E*
  then enact (the procedure bound to *P* with the value) .

Note that in the first equation, the type of the formal parameter denoted by the variable *T* does not appear in the action on the right-hand side of the semantic equation. This is because here we are only interested in the *dynamic* behaviour of the procedure declaration. The type of the formal parameter would be used in the corresponding semantic equation for the procedure declaration in the *static semantics* of NANO-Δ.

In general, the closure of an abstraction can be taken at any place from the point the abstraction is formed, up to the point that the abstraction is enacted. In most programming languages, only the point of abstraction and enactment are typically used. If the closure is applied at the point of abstraction, then the encapsulated action receives the bindings current at the point of declaration. If the closure is applied at the point of enactment, then the encapsulated action receives the bindings current at the point of call. The first is consistent with *statically-bound* programming languages, and the latter is consistent with *dynamically-bound* programming languages. In NANO-Δ, the closure is formed at the point of declaration, therefore NANO-Δ is statically bound.

2.3.8 Hybrid Action Notation

Hybrid action notation is concerned with actions that use more than one facet. For example, the "allocate" action uses both the imperative and the functional facets.

The action "allocate *S*" allocates a new cell *c* of sort *S*, it then gives the cell *c* as a single transient labelled 0. *S* may be a subsort of cell which can be used to restrict the sort of datum that can be stored in *c*. For example, *S* may be cell[integer] which restricts *c* to cells that can only contain integer values.

The "allocate" action is used to specify the semantics of a NANO-Δ variable declaration:

- elaborate [["var" *I*:Identifier ":" *T*:Type]] =
  allocate a cell then bind *I* to the cell .

2.4 Natural Semantics of ACTRESS Action Notation

The natural semantics of ACTRESS action notation is given in Appendix A. We use this formal semantics of ACTRESS action notation to construct the proof of soundness for our sort inference algorithm. The proof is given in Chapter 7.

The natural semantics specifies rules for mapping the input transients, bindings, and store of an action (or yielder) to an outcome (or datum). The outcome of an action consists of the termination status of the action (*completed*, *failed*, or *diverged*) along with the output transients, bindings and store where appropriate. (A failed action may not produce transients or bindings.)

The natural semantics of ACTRESS action notation was developed by Moura and Watt, and is based on Mosses' operational semantics of standard action notation. We can use a natural semantics since we are ignoring the possibility of interleaving the performance of actions.

2.5 Example Language Specification: NANO-Δ

In the following sections, we present the complete specification of an example programming language NANO-Δ. This language is a subset of the Δ programming language[2] used by Watt in [Wat91,Wat93].

NANO-Δ is a small, imperative programming language with Pascal-like syntax. It has assignment, while, skip, block and procedure-call statements; constant, variable and simple procedure declarations; literal, identifier, addition and negation expressions; and boolean and integer types.

2. pronounced “triangle”

The specification consists of two parts: the abstract syntax of NANO-Δ and the dynamic semantics of NANO-Δ. The dynamic semantics is further divided into the semantic functions and the semantic entities.

2.5.1 Abstract Syntax

In this section we present the full abstract syntax of NANO-Δ. It uses the standard data notation for strings, tuples, and trees. It also provides a good example of using a data operation applied to arguments which are sorts to produce a result which is a sort. For example, the operation "[[_ _ _]]" is applied to the sort Statement, the individual ":=" and the sort Expression to give one of the subsorts of Statement. In a grammar specification, the join operation is used as the choice operator in ordinary BNF.

NANO-Δ Abstract Syntax

```
grammar:

•  Statement =    [[ Statement ";" Statement ]] | [[ "skip" ]] |
                  [[ Identifier ":=" Expression ]] |
                  [[ "while" Expression "do" Statement ]] |
                  [[ "let" Declaration "in" Statement ]] |
                  [[ Identifier "(" Expression ")" ]] .

•  Expression =   Literal | Identifier | [[ "–" Expression ]] |
                  [[ Expression "+" Expression ]].

•  Declaration =  [[ "const" Identifier "~" Expression ]] |
                  [[ "var" Identifier ":" Type ]] |
                  [[ "proc" Identifier "(" Identifier ":" Type ")"  "~" Statement ]] |
                  [[ Declaration ";" Declaration ]] .

•  Type =         "int" | "bool" .
```

2.5.2 Dynamic Semantics

This section contains the complete dynamic semantics of NANO-Δ. It is composed of the semantic entities and the semantic functions.

2.5.2.1 Semantic Entities

This section contains the specification of the NANO-Δ semantic entities. They are very straightforward since NANO-Δ does not include any complicated sorts of data.

NANO-Δ Semantic Entities

needs: Data Notation .

introduces: value, procedure .

(1) value = integer | truth-value .

(2) procedure = abstraction[storing|diverging][using the value|current storage] .

(3) storable = value .

(4) bindable = value | cell | procedure .

2.5.2.2 Semantic Functions

The semantic functions introduced by NANO-Δ are evaluate, execute, and elaborate. These map expressions, statements, and declarations respectively to the actions denoting their meaning.

The specification of the semantic functions requires the standard specification of action notation, as well as the specifications of abstract syntax and semantic entities for NANO-Δ given in the previous two sections.

NANO-Δ Semantic Functions

needs: Action Notation, NANO-Δ Abstract Syntax, NANO-Δ Semantic Entities.

introduces: evaluate _, execute _, elaborate _ .

- evaluate _ :: Expression → action[giving a value]
  [using current bindings|current storage] .

(5) evaluate L:Literal = give valuation of L .

(6) evaluate I:Identifier =
    give the value bound to I
    or give the value stored in the cell bound to I .

(7) evaluate [["–" E:Expression]] =
    evaluate E then give negation(the integer) .

(8) evaluate [[E_1:Expression "+" E_2:Expression]] =
    | | evaluate E_1 then give the integer label #1

```
     | and
     | | evaluate E2 then give the integer label #2
     then give sum (the integer#1, the integer#2) .
```

- execute _ :: Statement → action[storing|diverging]
      [using current bindings|current storage] .

```
(9)  execute [[ "skip" ]] = complete .

(10) execute [[ S1:Statement ";" S2:Statement ]] =
         execute S1 and then execute S2 .

(11) execute [[ I:Identifier ":=" E:Expression ]] =
         evaluate E then store the value in the cell bound to I .

(12) execute [[ "while" E:Expression "do" S:Statement ]] =
         unfolding
         | | evaluate E
         | then
         | | | execute S and then unfold
         | | else complete .

(13) execute [[ "let" D:Declaration "in" S:Statement ]] =
         | furthermore elaborate D
         hence execute S .

(14) execute [[ P:Identifier "(" E:Expression ")" ]] =
         | evaluate E
         then enact (the procedure bound to P with the value) .
```

- elaborate _ :: Declaration → action[storing|binding]
      [using current bindings|current storage] .

```
(15) elaborate [[ "const" I:Identifier "~" E:Expression ]] =
         evaluate E then bind I to the value .

(16) elaborate [[ "var" I:Identifier ":" T:Type ]] =
         | allocate a cell
         then bind I to the cell .

(17) elaborate [[ "proc" P:Identifier "(" A:Identifier ":" T:Type ")" "~" S:Statement]] =
         bind P to closure abstraction
         | | furthermore bind A to the value
         | hence execute S .

(18) elaborate [[ D1:Declaration ";" D2:Declaration ]] =
         elaborate D1 before elaborate D2 .
```

Chapter 3

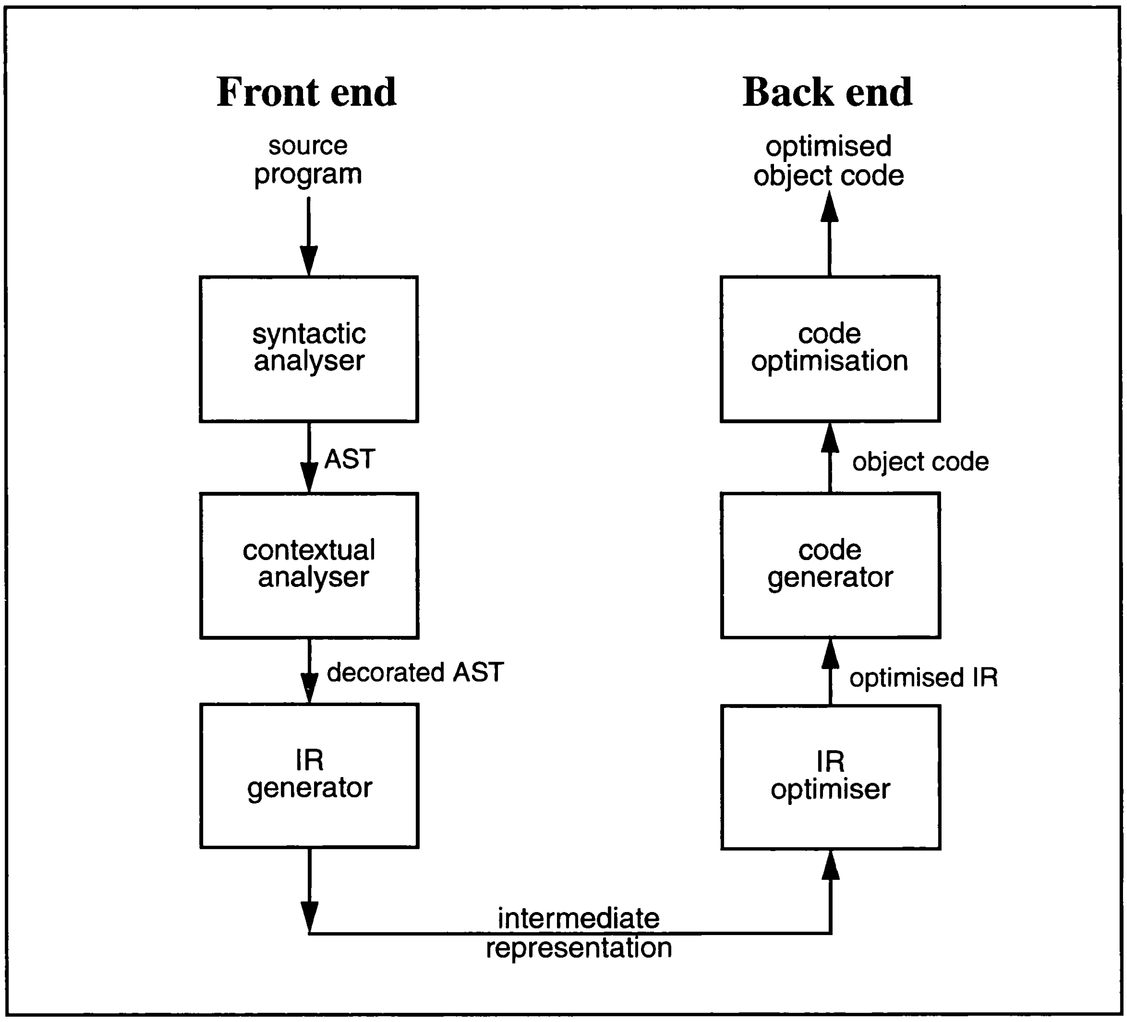
An Overview of Semantics-directed Compiler Generation

3.1 Introduction

The development of high-level programming languages in the 1950s revolutionised programming. A key component of this breakthrough was the development of robust and efficient compilers. A *compiler* is a software tool that translates a program expressed in a high-level language to the equivalent program expressed in a machine language. A *correct* compiler must preserve the exact meaning of a program during translation. A compiler for a modern high-level language, such as Ada or C++, is an extremely complex piece of software, taking many person-years to develop, and consisting of many 100,000s of lines of code. Unfortunately, few hand-written compilers are ever correct.

Initially, each new programming language or new target machine required an entirely new compiler to be written from scratch. Now, however, most compilers are split into a language-dependent *front-end*, and a machine-dependent *back-end*. The structure of such a typical compiler is given in Figure 3.1. The front-end and back-end of the compiler communicate through an *intermediate representation* (IR) of the program. A new programming language will require the development of a new front-end for the compiler, and a new target machine will require the development of a

Figure 3.1: The structure of a typical optimising compiler



new back-end. There are, however, still problems if the new programming language or target machine introduces a change in the intermediate representation, as this will require the modification of all of the existing front- and back-ends. However, even with such a division of responsibility in the compiler between the programming language and the target machine, developing a compiler still requires a substantial amount of time and effort.

To ease the development of a compiler for new programming language or a new target machine, a number of *ad hoc* systems have been produced that assist in the production of one or more phases of the compilation process. We refer to these as *compiler writing systems*, and they are essentially just a collection of tools used by the compiler writer. We briefly consider some examples of such systems in Section 3.2. Whilst these systems are a great help in the production of a compiler, they are of limited use in checking the correctness of the compiler. The descriptions used in the generation process bear little resemblance to any formal specification of the language, and any errors in the descriptions will lead to an incorrect implementation.

There have also been a number of attempts to produce systems which can generate a complete compiler given a *formal* specification of a programming language. We refer to these as *semantics-directed compiler generation systems*, and they are meant to be used by the language designer directly. Such systems differ in the mathematical formalism used to describe the syntax and semantics of the programming language, and the approach taken to generate the compiler. Since a semantics-directed compiler generator actually processes the formal specification of the programming language, the correctness of the generated compiler can be proven [Mei92,Pal92b].

In Sections 3.3 to 3.6, we consider some of the semantics-directed compiler generation systems that have been developed, and the approaches that have been used. We classify the systems according to the semantics methodology used, and consider systems based on denotational semantics, high-level semantics, and action semantics.

3.2 Compiler Writing Systems

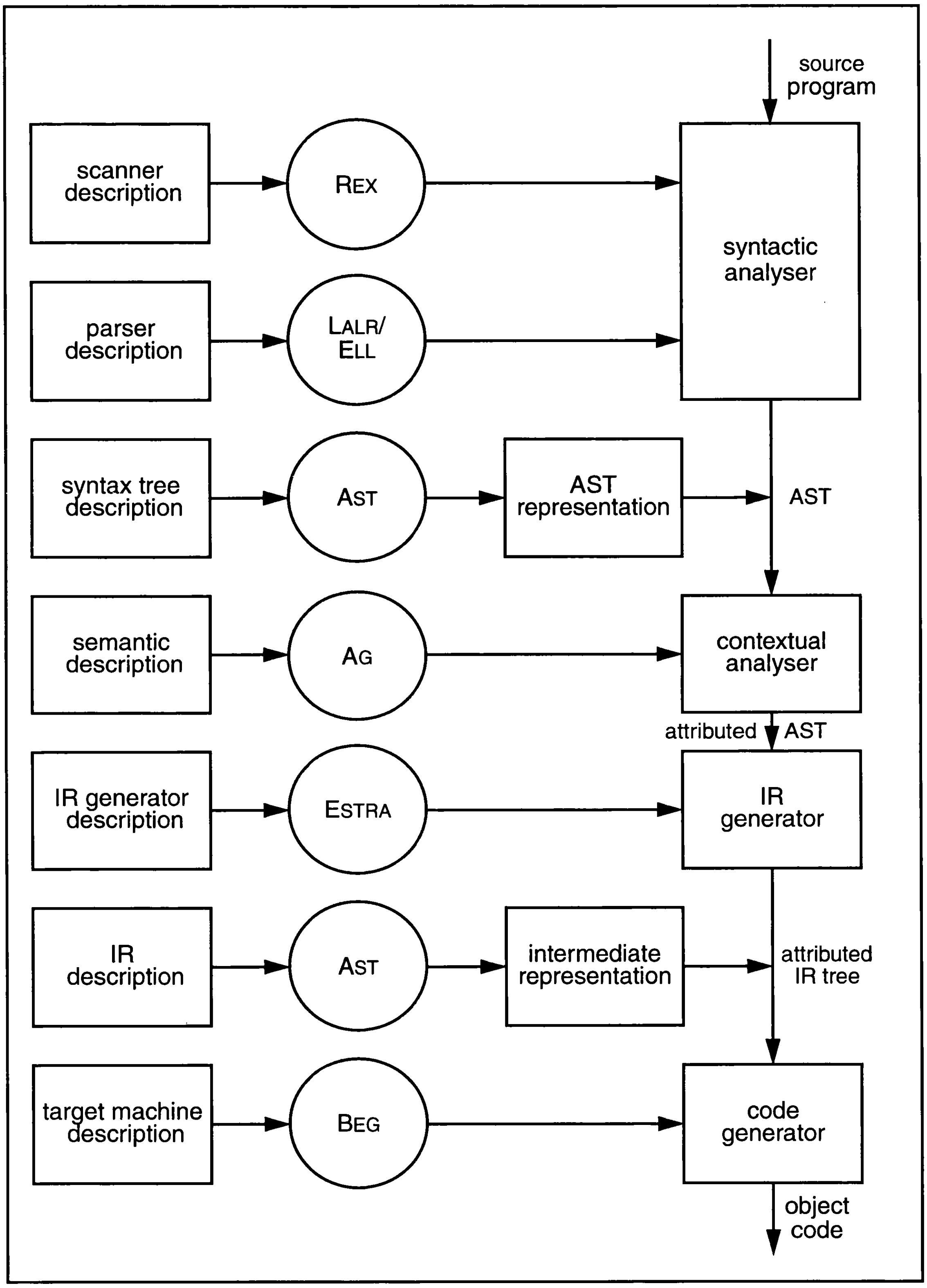
Several systems have been produced that ease the task of writing a compiler. Such systems offer tools to produce one or more parts of the compiler from appropriate descriptions written by the compiler-writer. The most widely-used tools are LEX[LS75], for generating lexical analysers, and YACC[Joh75], for generating LALR(1) parsers. These tools were originally developed for the UNIX operating system, but there are now versions available (some free) for virtually every development platform.

More recently, systems have been developed that provide tools to generate an entire compiler. For example, two such systems are ELI[GHL+92,Kas93,Wai93] and COCKTAIL[ESL89,GE90]. Both are based on attribute grammars, and use a collection of tools to produce the different phases of a compiler. To illustrate the structure of a typical compiler writing system, we will consider the COCKTAIL system in more detail.

The COCKTAIL system consists of seven separate tools for generating the phases of a compiler (and other related types of processing tool). The structure of the COCKTAIL system is shown in Figure 3.2, and contains the following components:

- REX constructs a lexical analyser from a scanner description written in regular expressions. (REX is much like LEX.)
- LALR constructs an LALR(1) parser from a parser description written in EBNF notation. The description may contain semantic actions to be executed when particular rules are reduced. (LALR is much like YACC.)
- ELL constructs an LL(1) (recursive descent) parser from a similar EBNF parser description. Again, the description can contain semantic actions.
- AST generates an abstract data type definition for attributed trees. It can be used to generate the AST representation used in the front-end of the compiler, or the intermediate representation used between the front-end and back-end of the generated compiler.
- AG generates an *attribute evaluator* from an *ordered attribute grammar*. The attribute evaluator traverses the input tree calculating the values of the attributes. The order of the traversal is determined by the dependencies between the attribute values. The attribute evaluator is used to perform contextual analysis where the attributes contain type information.
- ESTRA generates a tree transformer for an attributed tree. The generated

Figure 3.2: The structure of the COCKTAIL system



transformer can change the input tree into an arbitrary type of output. The transformation is described by a set of rules, or patterns, that are matched against the input tree. Each pattern has an associated action that is executed, when it is matched, to generate the output. A tree transformer can either be used to translate the AST into the intermediate representation, or to optimise the intermediate representation.

- BEG generates a code selector and register allocator from a description of the target machine. Code selection is performed using pattern matching, where fragments of the input tree are mapped onto machine instructions. In the description, the machine instructions are annotated with their register requirements, for example, the allowable registers for an instruction, or any registers altered by it. This information, together with a description of the target machine's register set, is used to construct a register allocator.

All of the tools in the COCKTAIL system were originally written in MODULA-2, but they have also been automatically translated to C using a MODULA-to-C translator, itself generated by COCKTAIL. Also, most of the tools can express their generated module in either MODULA-2 or C.

3.3 Denotational Semantics

The classical denotational semantics of Scott and Strachey[Sch86,Sto77] uses functions written in λ-calculus to represent the mapping of programs to their meaning. Compiler generation techniques using denotational semantics have been studied the longest. This is partly due to the use of denotational semantics to describe real programming languages, and partly due to the close relationship between λ-calculus and declarative programming languages such as LISP, SCHEME, ML, and HASKELL which provide a mechanism for executing denotational specifications.

3.3.1 Classical Systems

The first semantics-directed compiler generators were developed by Mosses, Paulson, and Wand. They all used denotational semantics, and all adopted the same general approach to the problem. We will refer to these systems as the *Classical Systems.* In the next section, we discuss the method they employ, and then consider each of them in more detail in the following three sections.

3.3.1.1 General Technique

In the classical approach, the semantics of the language are used to generate a translator from the abstract syntax tree of a program to a large λ-expression. This expression is then reduced (simplified) using the laws of λ-calculus, and the simplified expression forms the "compiled" form of the program. When "run", the program's input can be supplied to the expression, and it can then be further reduced to produce the program's output. The reduction arises through repeated application of the β-reduction rule in λ-calculus.

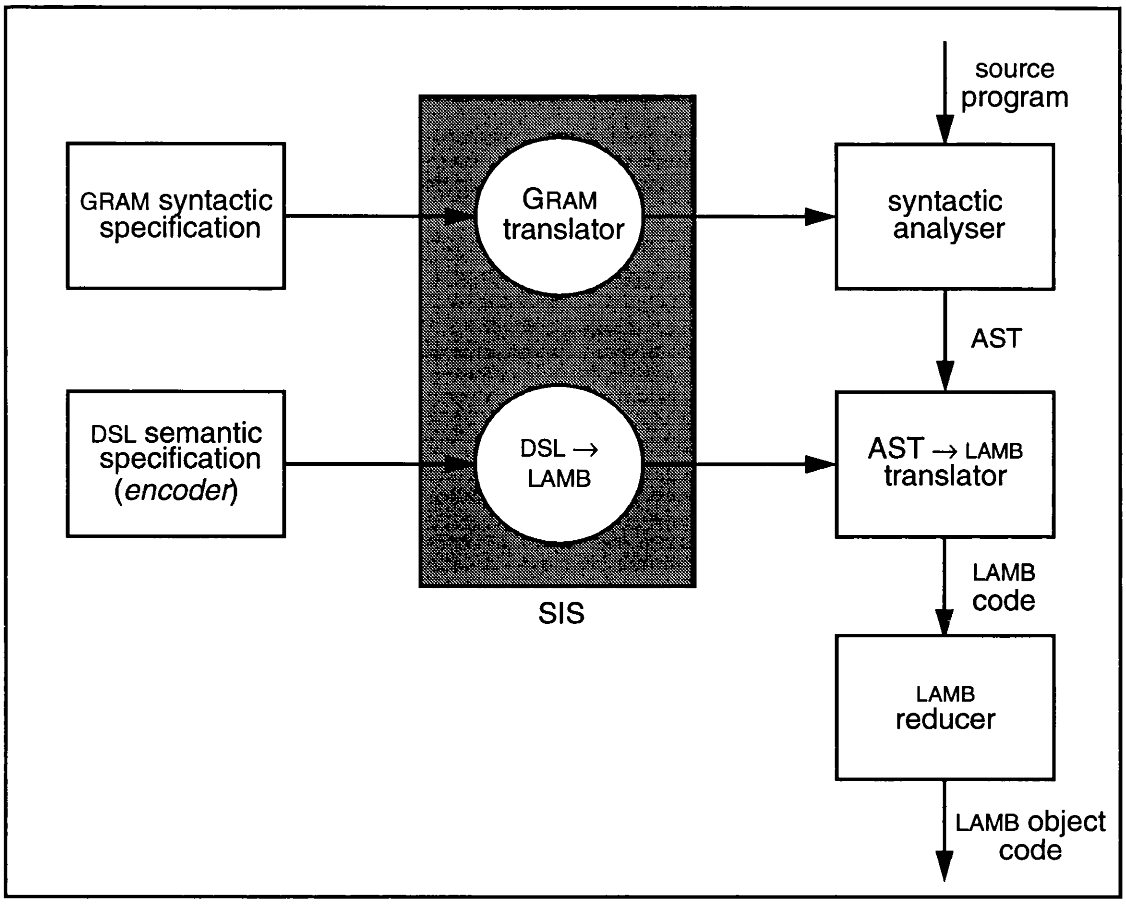
We can represent the compiler for a language $\mathcal{L}$, using the following equation:

$$compile_{\mathcal{L}} = reduce_{\lambda} \circ translate_{\mathcal{L} \to \lambda} \circ parse_{\mathcal{L}} \qquad \textbf{(3-1)}$$

where $parse_{\mathcal{L}}$ and $translate_{\mathcal{L} \to \lambda}$ must be generated from the language specification, but $reduce_{\lambda}$ is the same for all generated compilers.

The three classical systems differ in a number of ways. For example, the program may be represented in SCHEME rather than λ-calculus, the generated λ-expression may be compiled to abstract machine code before being executed, or the translator may not simplify the generated λ-term. These differences, however, do not greatly affect the overall performance of the system, or indeed, address the weaknesses of the classical approach outlined below.

Figure 3.3: The structure of SIS

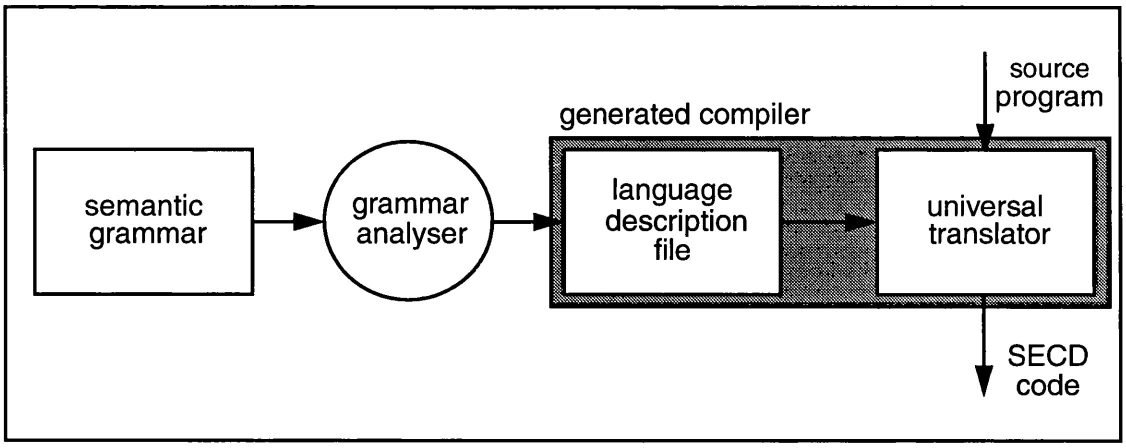


3.3.1.2 Mosses' Semantics Implementation System

Mosses was the first person to develop a system that translates a language specification into an compiler[Mos79]. His Semantics Implementation System, SIS, generates a compiler from a front-end specification and a semantic specification, called an *encoder.* The semantic specification is written in a notation called DSL, which uses a notation similar to λ-calculus, called LAMB, to specify the meaning of each program phrase. The front-end specification is written in a BNF-like notation called GRAM. The structure of SIS is shown in Figure 3.3.

SIS translates the GRAM specification to an SLR(1) parser that produces an abstract syntax tree (AST). The DSL specification is used to generate a translator from an AST to a LAMB expression. The AST → LAMB translator is also expressed in LAMB, and it is

Figure 3.4: The structure of the PSP system



applied to the AST of a program to produce a LAMB expression representing the compiled program. This expression is then reduced to normal form by a LAMB reducer, using a call-by-need strategy.

SIS is implemented in BCPL. It is extremely inefficient, requiring large amounts of processing time even for small specifications. Also, SIS lacks a type-checker for DSL, and this makes it extremely difficult to debug the semantic specification.

3.3.1.3 Paulson's Semantics Processor

Paulson's Semantics Processor (PSP) [Pau81] takes a *semantic grammar*, and produces a compiler that generates abstract machine code for a stack-based machine (SECD). A semantic grammar is a combination of an attribute grammar and a typed λ-calculus. The rules for calculating attribute values are written in λ-calculus. Designated attributes contain the semantics of the language. The system is implemented entirely in PASCAL. The structure of the PSP system is shown in Figure 3.4.

The semantic grammar is processed in two stages:

- First, a grammar analyser translates the specification to a language description file (LDF) that contains LALR(1) parse tables and attribute dependency information.

- Second, a universal translator reads the LDF and becomes a compiler for the language. (In the same way as a generic parser may read a set of parse tables and become a parser for a particular language.)

The generated compiler then processes the source program as follows:

- The source program is parsed, and a directed acyclic graph (DAG) of its attribute dependencies is constructed. The DAG encodes the λ-expression representing the program.
- This DAG is then simplified (mainly by β-reduction).
- Finally, the DAG is translated to code for the SECD abstract machine.

The program is then run by executing the object code using an SECD interpreter.

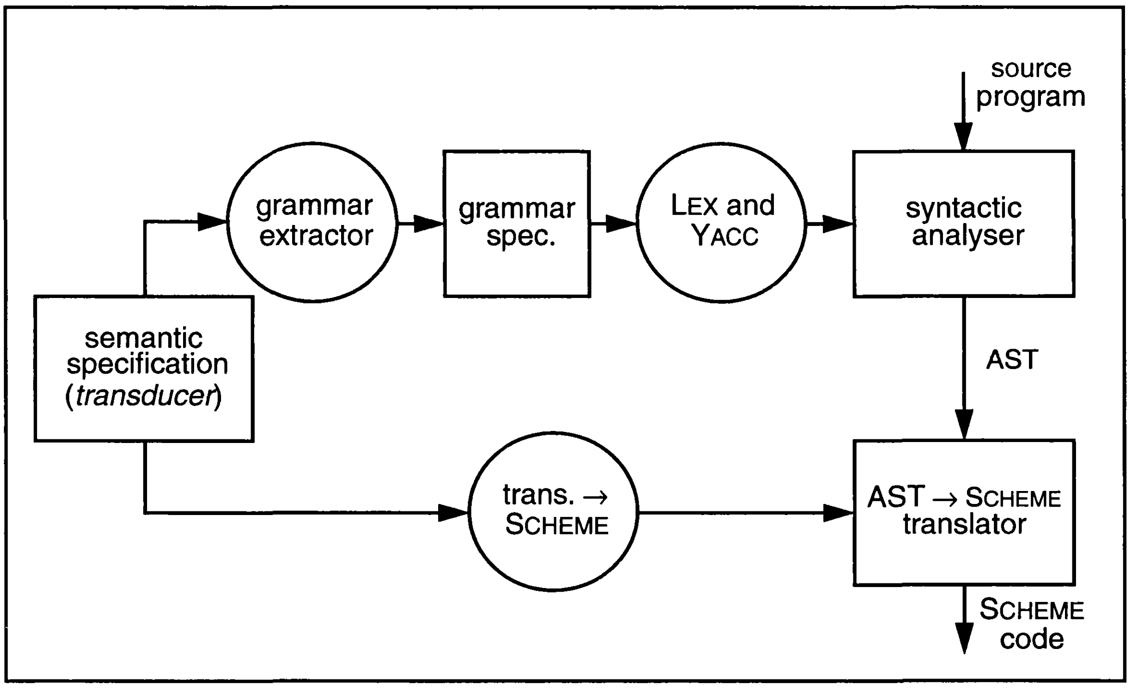
The generated compiler is very inefficient both in compile-time and the quality of the code produced. An experiment in generating a compiler for a subset of PASCAL, given in [BBK+82], states that the compiler was 25 times slower, and the generated code up to 1000 times slower than a hand-written PASCAL compiler.

3.3.1.4 Wand's Semantic Prototyping System

Wand's Semantic Prototyping System (SPS) [Wan82,Wan84] takes a semantic specification, called a *transducer*, and produces a compiler that generates SCHEME code. The transducer specifies a translation from an AST to SCHEME, and is written using LISP-like syntax. The structure of SPS is shown in Figure 3.5.

SPS processes the semantic specification twice. The first time, a grammar is extracted, and YACC and LEX input files for the parser are produced. The second time, the transducer is type-checked and translated into a SCHEME function that maps the source program AST into SCHEME code. The resulting SCHEME object code is executed by interpreting[1] it within the SCHEME 84 system. Unlike most other systems, SPS does

Figure 3.5: The structure of SPS



not reduce the generated SCHEME code before the program is executed.

SPS is implemented using several tools available within the UNIX environment, including FRANZ LISP, SCHEME 84, C, AWK, LEX, YACC and CSH.

3.3.1.5 Problems with the Classical Systems

The classical systems have several major flaws, which are discussed in detail in [Lee89], and summarised here:

- **Performance:** compilers generated using this approach typically produce object programs that execute three orders of magnitude slower than those from hand-written compilers.

- **Use of low-level notation:** since every aspect of denotational semantics must be represented using λ-abstraction and application, several inefficiencies arise. First, during compilation, a large proportion of the time is spent β-reducing the

[1] It could also be compiled with a SCHEME compiler.

generated λ-expression (often more than 60%). Second, during execution, the evaluation involves the manipulation of many closures.

- **Lack of resilience:** small changes in the source language that force changes in the underlying semantic domains require much, if not all of the existing semantics to be rewritten. For example, introducing jumps may require the change from direct to continuation passing style. In other words, the semantic functions are not resilient to changes in the semantic model being used.

- **Loss of semantic distinctions:** the "low-level" nature of the λ-calculus used to describe the semantics often obscures important differences in language features, representing them in the same way. This makes efficient compiler generation harder, or even impossible, if, for example, it has to attempt to detect the differences between variables and parameters, which are both represented in the environment of the program, but may be treated differently in the object code.

- **Over-specification of semantics:** classical denotational semantics typically makes it difficult to be imprecise about such things as the order of evaluation of sub-expressions. (In a good compiler, evaluation order is best determined by the complexity of the sub-expressions and not their syntactic ordering.) This "over specification" may force the compiler generator to adopt a particular strategy where a more flexible approach would be preferable.

3.3.2 Partial Evaluation

An alternative means of generating a compiler from a denotational semantic specification arises from the work on self-application and partial evaluation. There have been a number of different partial evaluators used to produce compilers. We begin by considering how partial evaluation is used to generate a compiler.

3.3.2.1 General Approach

A *partial evaluator* ($\mathcal{PE}_{\mathcal{L} \rightarrow \mathcal{L}}$) is a program that takes a program expressed in language $\mathcal{L}$, and some of its input, and returns the result of simplifying that program as much as possible with respect to the supplied input. In theory, a partial evaluator could be written for any language $\mathcal{L}$, but in practice, the best partial evaluators exist for the λ-calculus. Some examples of the partial evaluation of a λ-expression are:

- $\mathcal{PE}_{\lambda \rightarrow \lambda}\ (\lambda x.x)\ 6 = 6$

- $\mathcal{PE}_{\lambda \rightarrow \lambda}\ (\lambda\, (x, y)\, .\ \text{if}\ x < 0\ \text{then}\ x * y\ \text{else}\ x + y)\ (x, 0) =$
  $\lambda\, x.\ \text{if}\ x < 0\ \text{then}\ 0\ \text{else}\ x$

An interpreter for a programming language $\mathcal{L}$ ($interpret_{\mathcal{L}}$) is a function that takes a program expressed in language $\mathcal{L}$ ($P_{\mathcal{L}}$), and its input I, and executes $P_{\mathcal{L}}$ to produce its output O, i.e., we have:

- $interpret_{\mathcal{L}}\ P_{\mathcal{L}}\ I = O$

If a partial evaluator is applied to the interpreter, which must be expressed in λ-notation, and a program $P_{\mathcal{L}}$, then the resulting program can be executed directly with the program's input to give the output, i.e. it no longer requires the interpreter. We have:

- $\mathcal{PE}_{\lambda \rightarrow \lambda}\ interpret_{\mathcal{L}}\ P_{\mathcal{L}} = P_{\lambda}$

- $P_{\lambda}\ I = O$

Thus the application of the partial evaluator to the interpreter ($\mathcal{PE}_{\lambda \rightarrow \lambda}\ interpret_{\mathcal{L}}$) can be viewed as a compiler for $\mathcal{L}$. So partially evaluating an interpreter is equivalent to compiling. If the partial evaluator can take itself as an argument, i.e. it is *self-applicable*, then we can generate a compiler for $\mathcal{L}$ ($compile_{\mathcal{L}}$):

- $\mathcal{PE}_{\lambda \rightarrow \lambda}\ \mathcal{PE}_{\lambda \rightarrow \lambda}\ interpret_{\mathcal{L}} = compile_{\mathcal{L} \rightarrow \lambda}$

- $compile_{\mathcal{L} \to \lambda}\ P_{\mathcal{L}} = P_{\lambda}$
- $P_{\lambda}\ I = O$

The compiler no longer requires the partial evaluator in order to compile programs (in the same way as the compiled program no longer required the interpreter above).

Finally, if we partially evaluate the partial evaluator with respect to itself, we obtain a compiler generator ($generate_{\lambda \to \lambda}$):

- $\mathcal{PE}_{\lambda \to \lambda}\ \mathcal{PE}_{\lambda \to \lambda}\ \mathcal{PE}_{\lambda \to \lambda} = generate_{\lambda \to \lambda}$
- $generate_{\lambda \to \lambda}\ interpret_{\mathcal{L}} = compile_{\mathcal{L} \to \lambda}$

So a self-applicable partial evaluator can be used to construct a compiler generator that translates an interpreter into a compiler. Note, moreover, that a denotational semantic specification of a language $\mathcal{L}$ can be viewed as an interpreter for $\mathcal{L}$. Therefore, we have obtained a semantics-directed compiler generation system.

There have been a number of partial evaluators that have been used to generate compilers. The first partial evaluator, called MIX, was developed at the University of Copenhagen by Gomard and Jones[GJ91]. More recently, both Consel[Con88,Con93] and Bondorf[Bon91,Bon92] have developed partial evaluators called SCHISM and SIMILIX respectively. SIMILIX has been applied to the partial evaluation of an action notation interpreter[BP93], and is discussed in Section 3.5.2. In the next section, we will use SCHISM to illustrate the operation of a typical partial evaluator.

3.3.2.2 Consel's Schism

SCHISM[Con88,Con93] is a partial evaluator for pure applicative languages. It is designed as a *back-end* partial evaluator, i.e. it processes a core language that is sufficiently general to capture a variety of applicative languages, including pure subsets of SCHEME and ML.

SCHISM processes a source program in a number of phases:

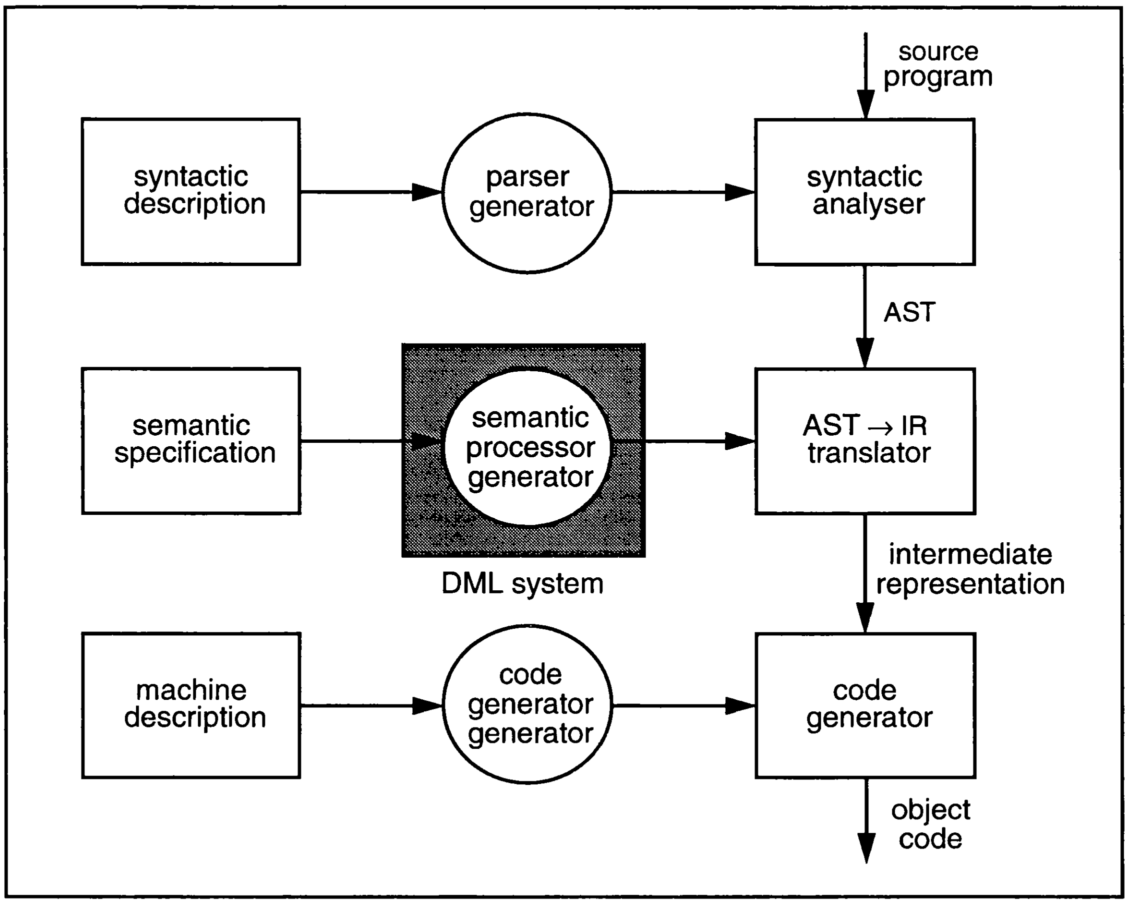
- First, all of the user-defined functions in the source program must be annotated with a *filter.* A filter tells the partial evaluator how to transform a function call, and how to propagate static arguments. Filters can be written by hand, or automatically generated by the system.
- Second, the source program undergoes *binding-time analysis.* Binding-time analysis determines which values in the program are static and which are dynamic. The binding-time analyser takes the source program (including filters), and a *binding-time description* of the program's input, i.e. a description of which parts of the inputs are static and which parts are dynamic. It produces a binding-time description of the entire program.
- Third, the source program undergoes *specialization.* This transforms the program using the binding-time information, and the known input values (or specialization values). The result of specialization is the *residual program.*
- Finally, the residual program is *re-sugared* to produce a human-readable version of the partially-evaluated source program.

SCHISM is written in SCHEME, and is self-applicable.

3.3.3 Denotational Meta-language

Petersson and Fritzson[PF92] have developed a compiler generator that uses a specification written in their denotational meta-language (DML). DML is a superset of STANDARD ML, and so uses notation similar to λ-calculus. The DML system consists only of a semantic processor generator. This accepts a DML specification, and produces one component of the resulting compiler, namely a translator from an AST to an intermediate representation that uses quadruples. A DML specification is written in the continuation-passing style. The AST-to-IR translator has been designed to interface

Figure 3.6: The structure of DML



with modules generated by existing parser generators and code generator generators. Alternatively, for the code generator, the quadruples can be expressed directly in C, and then compiled using an optimising C compiler.

The semantic processor generator is implemented in SCHEME, and initially produces an AST-to-IR translator also expressed in SCHEME. The AST-to-IR translator is then translated to C using a SCHEME-to-C translator. The AST-to-IR translator first translates the AST to a λ-expression, and then translates this λ-expression to produce quadruples. The most important part of the DML system is an efficient translation algorithm from λ-calculus to quadruples.

At this time, DML has only been used for a small C-like language, TINY-C. They illustrate that the object code for one program is comparable with the object code expected from an hand-written optimising compiler, but they give no results for either the compiler-generation time or the compile-time. Also, the efficient code generation algorithm relies on the DML specification using a small number of primitive operations, which can be mapped directly to quadruples.

3.3.4 Modular Denotational Semantics

Recently, work in functional programming has concentrated on the use of monads[Mog90,Wad90] to provide additional structure to the normal λ-calculus and allow the development of models for functional I/O and hidden state. Liang and Hudak[LH96] have adapted this work on monads to structure the denotational semantics of programming languages.

In their work, Liang and Hudak use a number of different *monad transformers* to model the various types of information used by a programming language, e.g. flow of control, environments, and storage. In this context, a monad transformer acts much like a facet in action semantics. The monad transformers act independently of one another, and a new transformer can be added without affecting the existing transformers or requiring the existing semantics to be re-written.

In addition, using monads allows the use of the monad laws to transform programs in safe ways. This allows an intermediate λ-expression to be simplified before code generation, in particular eliminating the monad transformer for environments so that bindings are not present at run-time. Finally, implementing the primitive monadic operators in a given target language provides an efficient method of code generation.

Although Liang and Hudak argue that their modular denotational semantics would provide an efficient basis for an automatic compiler generation system, no such system exists at present to provide results about the performance of this approach.

3.4 High-level Semantics

High-level semantics is a form of denotational semantics developed by Lee and Pleban[LP87,Lee89], and it uses a notation similar to the programming language STANDARD ML, with more or less syntactic sugar. It divides the semantics into two parts, the so-called *macro-semantics* and the *micro-semantics*. The macro-semantics (or simply semantics) forms the specification of the programming language. The micro-semantics forms the specification of a semantic model. The macro-semantics uses the operations provided by the micro-semantics, but is insulated from any changes in the semantic model that do not affect the signatures of the operations. However, if any changes are made to the signatures, then modifications to the macro-semantics are necessary. Fortunately, the parts of the macro-semantics affected are easily identified.

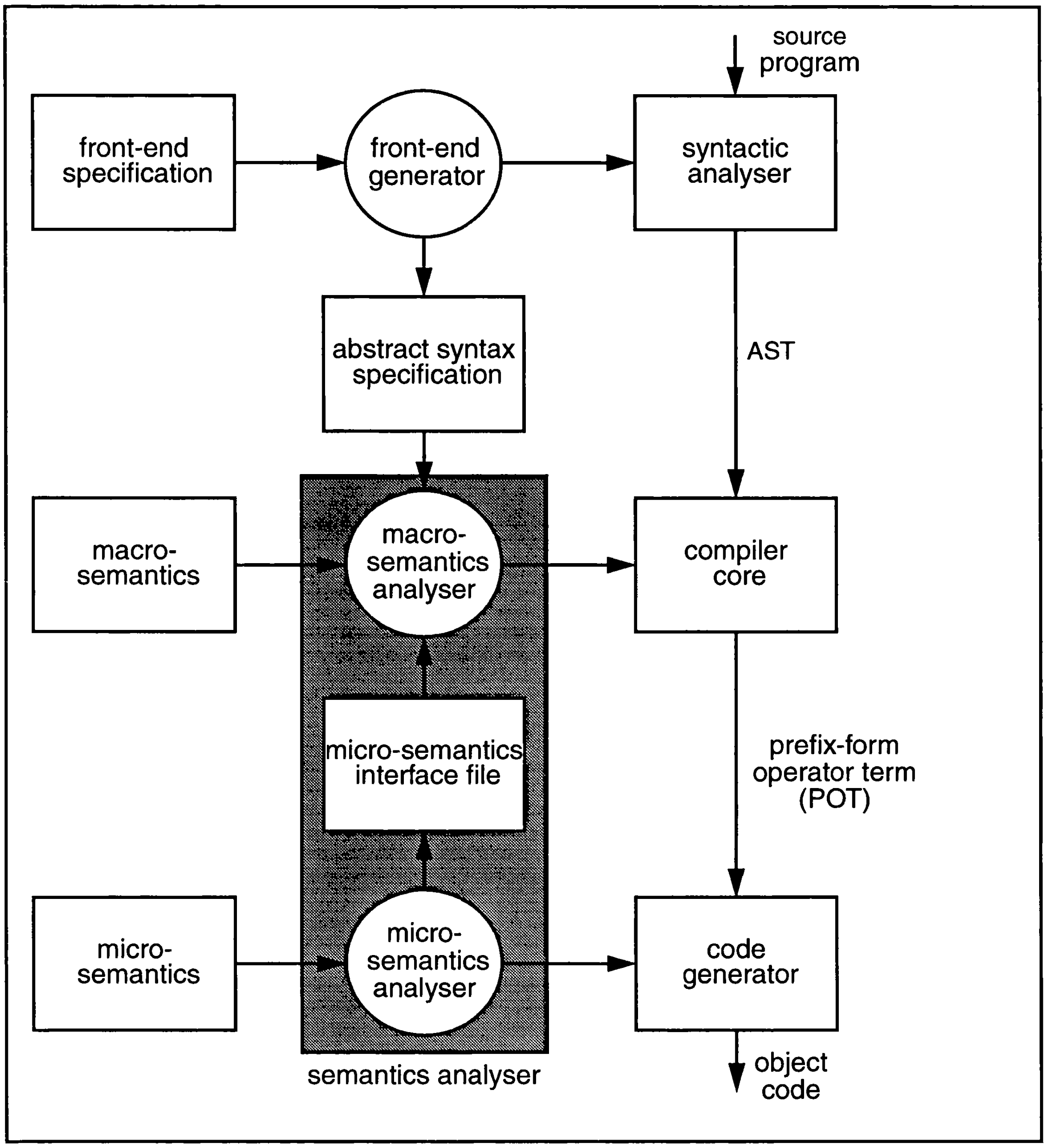
The separation of the language specification into a macro-semantics and a micro-semantics was inspired by the early work on action semantics, where the micro-semantics can be viewed as the specification of action notation itself. The micro-semantics specification, however, is not fixed, and so its algebraic properties cannot be predicted. Indeed, the micro-semantics specification is typically at a low level, forming either the specification of an abstract machine or of a code generator.

As well as developing high-level semantics, Lee and Pleban tested its use in semantics-directed compiler generation. Lee produced a system, called MESS, which is discussed below.

3.4.1 Lee's Mess

The MESS system automatically generates a single-pass compiler from the source language's syntax and macro-semantics, and a micro-semantic specification of either an abstract machine or a code generator. The MESS system is implemented in PASCAL and SCHEME on an IBM PC microcomputer, and it produces a compiler expressed in SCHEME. The structure of the MESS system is shown in Figure 3.7

Figure 3.7: The structure of MESS



MESS consists of a front-end generator, and a semantics analyser. The front-end generator is a straightforward parser generator, but it also produces a description of the abstract syntax that is required by the semantics analyser. The semantics analyser processes both the macro-semantics and the micro-semantics. The macro-semantics is used to produce a compiler core. The compiler core maps an AST to a term in the semantic model of the micro-semantics, that satisfies all of the static constraints of the macro-semantics. This term is called a *prefix-form operator term* (POT), since all of

Figure 3.8: Example specification for declarations in MESS

```
D: Decl -> ENV -> (ENV * DECL_ACTION)

D [[ decl ";" decls ]] env =
    let (env1, declAct1) = D [[ decl ]] env in
        let (env2, declAct2) = D [[ decls ]] env1 in
            (env2, DeclSeq (declAct1, declAct2))
        end
    end.

D [[ ]] env = (env, NullDecl).

D [[ "int" id ]] env =
    if notDeclared (id, env) then
        let b = currentBlockNumber (env).
            l = currentLevel (env).
            name = mkAlphaName (id, b).
            mode = varM ((name, int_type), l).
            newEnv = adAssoc (id, mode, b, env)
        in
            (newEnv, DeclSimpleVar (name, IntType))
        end
    else
        declError env [[ id ]]
            "Identifier already declared.".
```

the micro-semantic operations are prefix operations. The micro-semantics is used to produce a code generator, that translates the POT to object code.

As Lee reports [Lee89], the time taken to generate a compiler is considerable. Moreover, the compiler itself could be 10 times slower than a hand-written one. However, the code produced by the compiler would appear to be at least as good as a non-optimising commercial compiler.

Although high-level semantics, like action semantics, aims to be modular, readable, and separable, the notation used is not sufficiently abstract to concisely describe common programming constructs. For example, consider the example semantics for declarations taken from [LP87], and shown in Figure 3.8. It involves the explicit manipulation of compile-time entities such as variable modes (`mode`), block number and level (`currentBlockNumber` and `currentLevel`), and environments (`env` and `newEnv`). Micro-semantic operators such as `DeclSeq`, `NullDecl`, and `DeclSimpleVar` are analogous to the action notation "and then", "complete" and "bind" respectively.

3.5 Action Semantics

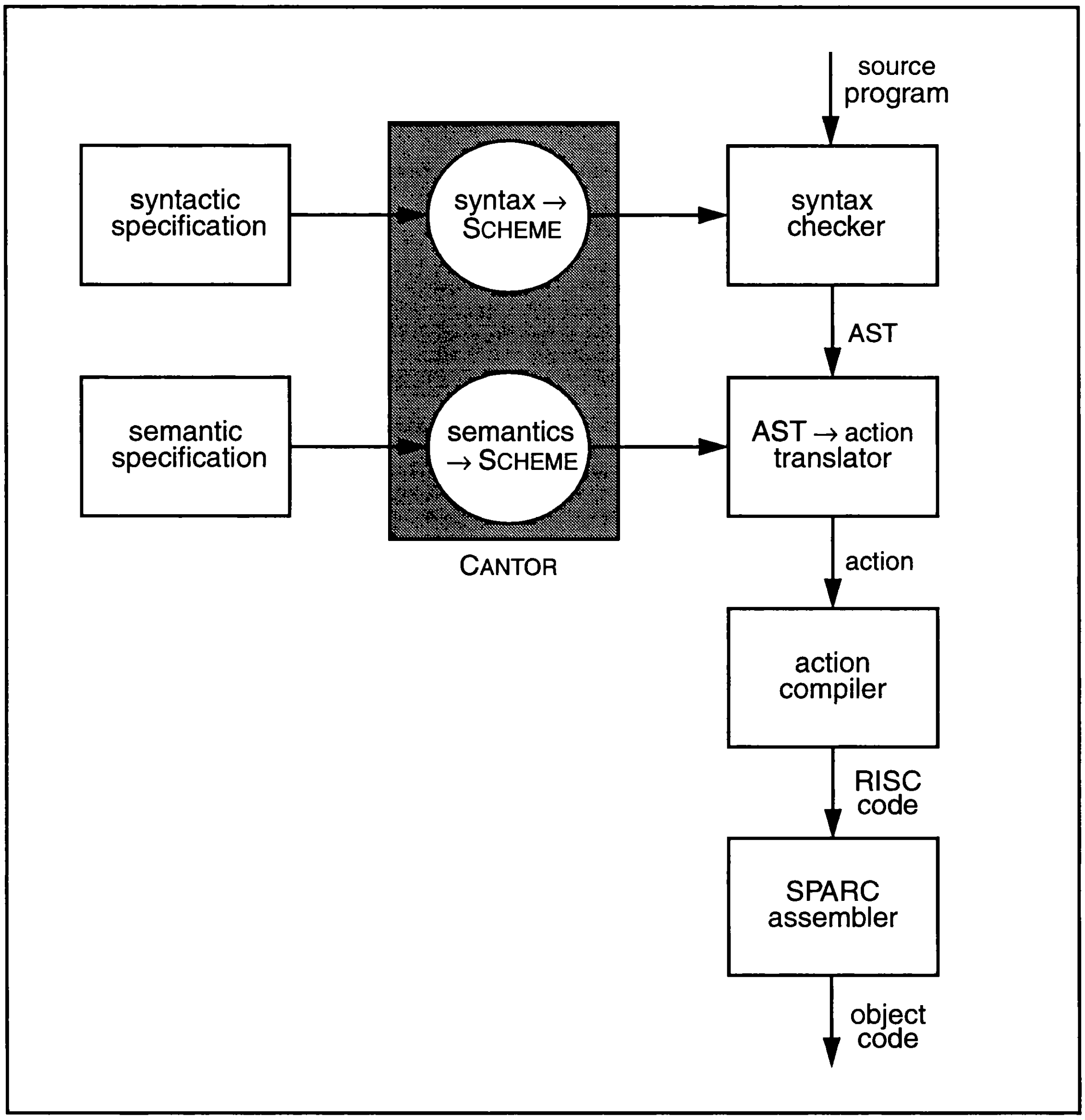
Apart from the ACTRESS system, which we discuss in detail in Chapter 4, there have been a number of other systems that use action semantics as the basis for a compiler generation system.

3.5.1 Palsberg's Cantor

Palsberg[Pal92b] has implemented a system called CANTOR, that generates a compiler that produces RISC assembly code for either the SPARC processor or the PA-RISC processor. The CANTOR system is written in PERL[WS91], and the generated compiler is expressed in SCHEME.

CANTOR accepts the same input file, written in LATEX, that is used to generate the formatted version of the language specification. From this input file, it identifies the action semantic modules for the syntax and semantic functions of the programming language. The syntactic specification is used to generate a *syntax checker.* The source program for a generated compiler must be written directly as an AST expressed in SCHEME. The syntax checker only verifies that this tree is well-formed with respect to the grammar of the language. The semantic specification is used to generate a translator from the AST to an action. This action is then sort checked and compiled by the action compiler to produce RISC assembly language. The sort checking guarantees that the action cannot fail when performed due to a sort error. (This allows the data used at run-time to be untagged.) The sort checking phase uses a simple top-down algorithm that calculates the sort of the outputs of the action given the sort of its inputs.

Figure 3.9: The structure of CANTOR



Importantly, Palsberg has proved the correctness of the action compiler used in the CANTOR system. Moreover, the compilation of actions and the RISC target language have been specified algebraically, and the correctness is proved solely within the algebraic framework.

Experiments with CANTOR[Pal92a] have shown that a generated compiler is about 300 times slower than a hand-written one, and that the object code is about 100 times slower than that produced by a hand-written compiler. Palsberg attributes these poor timings to a number of factors. Namely:

- The lack of compile-time constant propagation;
- Poor register allocation; and
- A naive representation of bindings, closures, and lists.

However, he states that improving these aspects of the action compiler would substantially complicate the proof of correctness.

3.5.2 Bondorf and Palsberg's system

After his experience with the CANTOR system, Palsberg then considered a different approach to constructing an action semantics directed compiler generator. In conjunction with Bondorf[BP93], he developed an action compiler using the technique of partial evaluation (as explained in Section 3.3.2). Using Bondorf's partial evaluator SIMILIX[Bon91,Bon92], they generate an action compiler by partially evaluating an action interpreter written in SCHEME. The action interpreter is systematically derived from the operational semantics of their subset of action notation, and they note that it is only about one-third of the size of the action compiler used in CANTOR.

The performance of a compiler generated with this system is better than one generated with CANTOR. In their experiments, the compiler was more than ten times faster. (Although they have to adjust their figures to compensate for differences in the SCHEME implementation used.) The compiler generation time, however, is an order of magnitude slower. This is acceptable, since a compiler is generated much less often than it is run. The generation time reflects the greater level of analysis required by the partial evaluator. The run-time performance of compiled programs is comparable to CANTOR, which is significant as CANTOR is designed to produce RISC object code, whereas the new system produces SCHEME object code. However, these timings required some hand annotations of the interpreter to convince SIMILIX that more of the compile-time data is static than its own analysis detects. Without these annotations, the object code was several times slower than that produced by CANTOR.

3.5.3 Ørbæk's OASIS

Ørbæk (who implemented the CANTOR system) has developed his own semantics-directed compiler generator called OASIS[Ørb93,Ørb94]. OASIS has the same overall structure as CANTOR. In OASIS, however, the action compiler has been replaced by one that generates better code, at the expense of the "provably correct" property.

The OASIS action compiler performs four different analyses of the input action. Of these, three are concerned entirely with code generation. The fourth, however, is a sort analysis of the action. The sort analysis is similar to the one used in CANTOR, although it has been extended to allow non-tail-recursive use of "unfold". It consists of four separate analyses: binding time analysis, constant propagation, commitment analysis, and termination analysis. The binding time analysis determines which values are known statically and which are known dynamically. Static values are propagated throughout the action, and expressions involving only static data are simplified. For example, the operation "sum (1, 2)" is replaced by "3". Dynamic values are placed in the store, and retrieved when they are used.

The performance of an OASIS generated compiler is good. The compiler is about two orders of magnitude faster than one produced by CANTOR, and the object code produced by the compiler is also about two orders of magnitude faster than the object code produced by a CANTOR-generated compiler. When compared with a hand-written compiler, however, the compile time is on average 6.5 times slower, and the object code is at most 4 times slower.

3.5.4 Doh's system

Most recently, Doh[Doh95] has presented another application of partially evaluating actions. He is concerned with producing an automatic action transformer that eliminates static computation in an action to leave a residual action containing only dynamic computation. This is an alternative approach to the action transformations proposed by Moura[Mou93b] and used in the ACTRESS system to simplify an action before code generation.

Doh has adapted the two-level type system used by Nielsen and Nielsen[NN92] to produce a two-level type inference for action notation. The two-levels of types distinguish between static and dynamic values, and an action or yielder with an entirely static type can usually be eliminated by partial evaluation.

Importantly, Doh has extended his two-level type inference to consider inferring the type of an action appearing in an action semantic specification. Whilst a complete type cannot normally be inferred for such an action, Doh's system uses type variables to construct an action that has been annotated with a type scheme. Certain errors in an action semantic specification can be detected at this stage, before compiler generation takes place. Finally, when an annotated action is generated for a given program, then its type can be calculated by composing the type annotations of its sub-actions and instantiating type variables, thereby eliminating the need to repeatedly infer the types of primitive actions.

Although Doh presents some important results, there is currently no implementation of his two-level type inference for actions or action semantic specifications.

3.6 Conclusion

Over the last twenty years, semantics-directed compiler generation has improved significantly. The early systems were slow both at compile-time and at run-time, typically three orders of magnitude slower than a hand-written compiler. The most recent systems have reduced this time penalty to one order of magnitude or better.

The aim of semantics-directed compiler generation is to produce an efficient compiler directly from the formal specification of a programming language. Although no system to date has been used to generate a compiler for a "real" programming language, substantial progress has been made towards this goal.

Early systems suffered due to the poor pragmatic qualities of the semantic specifications used—it was just too hard to generate efficient code from such specifications. More recent systems have used either a modified version of denotational semantics, or an alternative framework, such as action semantics, to produce efficient code. The quality of the code produced is no longer a drawback to using such systems to generate "realistic" compilers.

The next breakthrough in semantics-directed compiler generation will involve producing systems robust enough to process the entire semantics of real-world programming languages. Such systems could genuinely be used as part of the programming language life-cycle.

Chapter 4

ACTRESS: an Action Semantics Directed Compiler Generator

4.1 Introduction

The ACTRESS compiler generation system[BMW92a,BMW92b] consists of a number of modules written in SML and a tool, called the *actioneer generator*, for creating a language-specific module called an *actioneer*. Some of these modules are shown in Figure 4.1. The role of each module can be summarised as follows:

- The **action notation parser** (*parse*$_{\mathcal{A}}$) parses an input action in textual form and produces its corresponding abstract syntax tree (*action tree*).

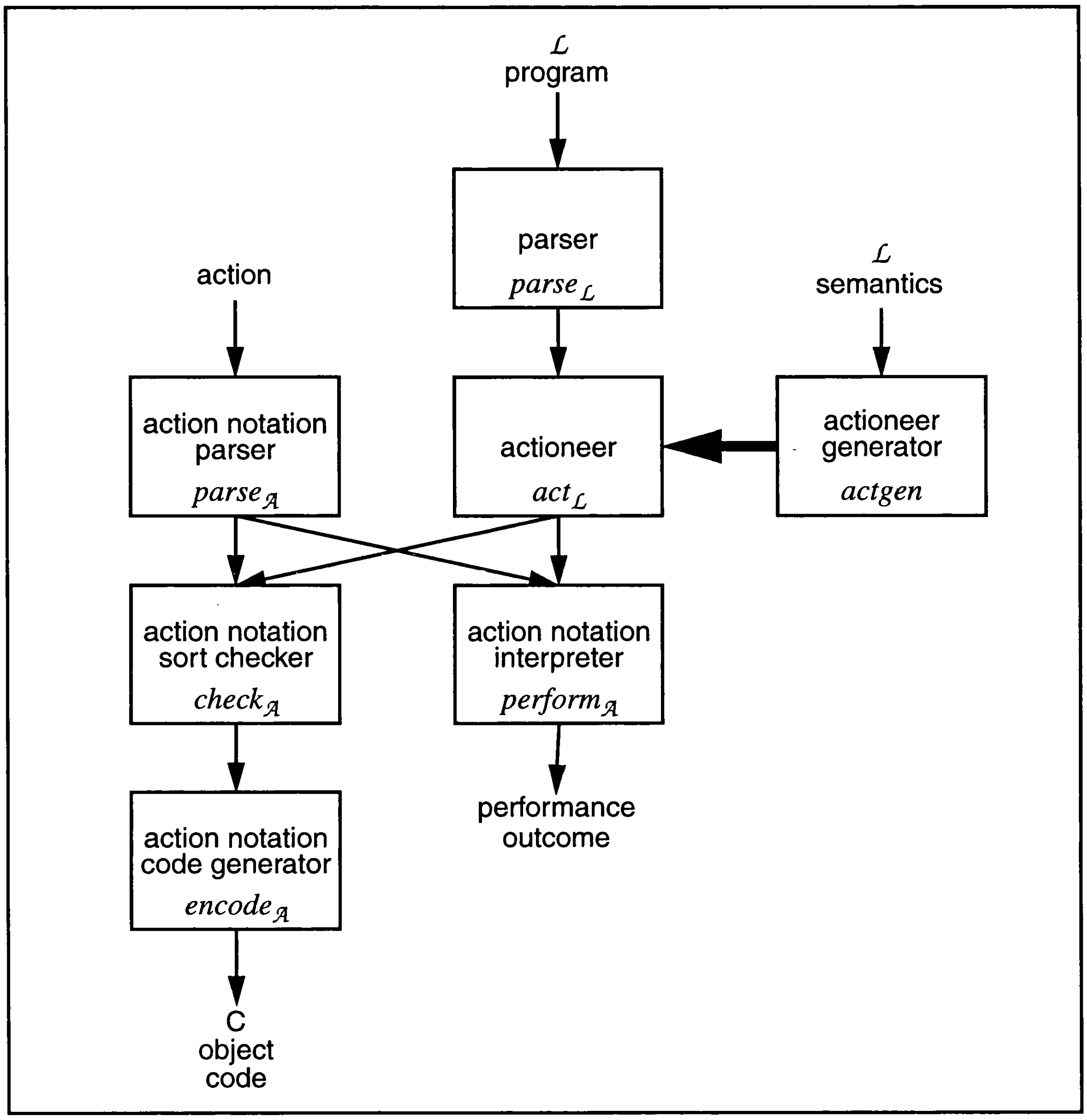
- The **action notation sort checker** (*check*$_{\mathcal{A}}$) infers the sort information for the transients and bindings in the given action tree, and checks that this information is used consistently. It produces the action tree decorated with the inferred sort information.

- The **action notation code generator** (*encode*$_{\mathcal{A}}$) takes a decorated action tree and generates object code expressed in (low-level) C.

- The **action notation interpreter** (*perform*$_{\mathcal{A}}$) directly interprets an action tree and gives the outcome of performing it.

- An **actioneer** (*act*$_{\mathcal{L}}$) takes a program in language $\mathcal{L}$ expressed as an abstract

Figure 4.1: The structure of ACTRESS



syntax tree (AST), and translates it to the corresponding action tree representing that program given the action semantics of $\mathcal{L}$.

- The **actioneer generator** (*actgen*) takes an action semantic specification for a language $\mathcal{L}$ and generates an *actioneer* for $\mathcal{L}$.

As can be seen from Figure 4.1, several tools can be constructed by composing these modules in different ways. For example, by composing the action notation parser with the action notation interpreter, we get a simple method of interpreting action terms:

$$interpret_{\mathcal{A}} = perform_{\mathcal{A}} \circ parse_{\mathcal{A}} \qquad \textbf{(4-1)}$$

Alternatively, by composing the action notation parser, sort checker and code generator, we get a compiler for action notation:

$$compile_{\mathcal{A}} = encode_{\mathcal{A}} \circ check_{\mathcal{A}} \circ parse_{\mathcal{A}} \qquad \textbf{(4-2)}$$

Using the actioneer generator, we can produce an actioneer for a programming language $\mathcal{L}$, which can be similarly composed with the action notation interpreter and a parser for $\mathcal{L}$ ($parse_{\mathcal{L}}$) to give an interpreter for $\mathcal{L}$:

$$interpret_{\mathcal{L}} = perform_{\mathcal{A}} \circ act_{\mathcal{L}} \circ parse_{\mathcal{L}} \qquad \textbf{(4-3)}$$

Most important of all is, of course, composing an $\mathcal{L}$ parser with an $\mathcal{L}$ actioneer and the action notation sort checker and code generator to produce a compiler for $\mathcal{L}$:

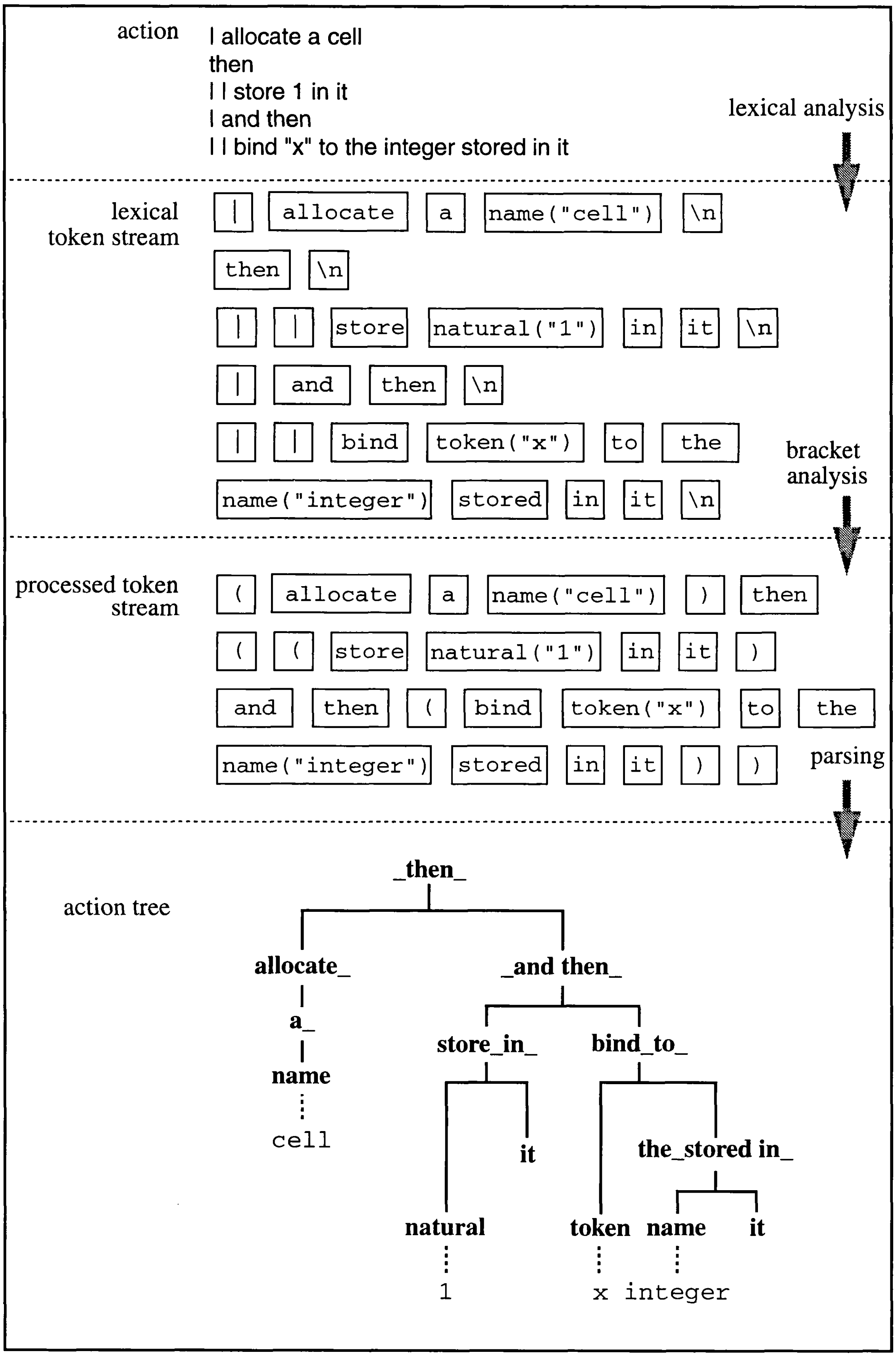
$$compile_{\mathcal{L}} = encode_{\mathcal{A}} \circ check_{\mathcal{A}} \circ act_{\mathcal{L}} \circ parse_{\mathcal{L}} \qquad \textbf{(4-4)}$$

The action notation parser and sort checker were implemented by the present author. The action notation code generator and interpreter were implemented by Moura[Mou93a, Mou93b]. The actioneer generator uses an extended version of the action notation parser implemented by the present author, and a translation phase implemented by Moura. The operation of each of the ACTRESS modules is explained in more detail in the following sections.

4.2 Action Notation Parser

The action notation parser reads an ASCII representation of an action, parses it, and generates the corresponding action abstract syntax tree (*action tree*). The action notation parser consists of three sub-phases: lexical analysis, bracket analysis, and parsing. It is partly constructed using the lexical analyser generator ML-LEX[AMT94], and the parser generator ML-YACC[TA90], provided with New Jersey ML. Figure 4.2

Figure 4.2: An example of parsing an action


action
| allocate a cell
then
| | store 1 in it
| and then
| | bind "x" to the integer stored in it
lexical analysis
lexical token stream
| allocate a name("cell") \n
then \n
| | store natural("1") in it \n
| and then \n
| | bind token("x") to the
name("integer") stored in it \n
bracket analysis
processed token stream
(allocate a name("cell")) then
((store natural("1") in it)
and then (bind token("x") to the
name("integer") stored in it))
parsing
action tree
then
allocate_
a_
name
cell
and then
store_in_
natural
1
it
bind_to_
token
x
the_stored in_
name
integer
it

illustrates the lexical analysis, bracket analysis, and parsing of an example action.

The ASCII representation of an action is very simple. There are only two main difficulties in representing arbitrary actions. First, the vertical rules normally used to show grouping must be represented. This is done by constructing a rule with a column of ASCII pipe characters "`|`". Note that, for simplicity, the parser is not concerned with the precise vertical alignment of the pipe characters between lines, but merely with the number of pipe characters present at the start of the line. Second, the parser must restrict the syntax for data operations to allow a simple parsing algorithm to be used. As a result, data operations must be of the form "$O(T_1, \ldots, T_n)$" for an operation symbol O and argument terms $T_1, \ldots, T_n$, as opposed to the "mix-fix" syntax allowed in standard data notation.

Lexical analysis is straightforward. The action notation symbols are treated as reserved words, where each word has its own lexical token. For example, the action notation symbol "and then" is interpreted as the lexical token "`and`" followed by the lexical token "`then`". There are lexical tokens for punctuation such as "#", "`|`", "`[`" and "`]`". Finally, there are lexical tokens for names (for data constants, and data operations), tokens[1] (quoted strings), and natural literals. Each of these tokens contains an attribute for the corresponding spelling of the token. For example, `name("cell")`, `token("x")`, and `natural("1")` are all lexical tokens.

Bracket analysis is necessary to eliminate the vertical-bar notation used to indicate grouping. This notation is similar to the offside rule found in functional programming languages[Lan66]. Bracket analysis is best done before parsing to allow the use of a simple LALR parser. However, if the lexical analyser could be extended to use a stack of tokens, then bracket analysis could be integrated into the lexical anaylsis phase. For example, this is the approach taken in the Glasgow HASKELL compiler[Gla96]. Bracket

[1] i.e. there is a lexical token called "`token`".

analysis is performed as follows. The lexical analyser returns a token stream including tokens for the pipe character, "|", and the new-line character, "\n". The bracket analyser counts the pipe tokens at the start of a line and determines how many open-parenthesis or close-parenthesis tokens should be inserted. An increase in the number of pipe tokens requires open-parenthesis tokens to be inserted, and a decrease requires close-parenthesis tokens to be inserted. The number of tokens inserted is determined by the difference in the number of pipe tokens at the start of the current line, and the number at the start of the previous line. Token insertion occurs when the first non-pipe token on the line is received. The pipe tokens and new-line tokens are deleted by the bracket analysis, and the token stream received by the parser contains left and right-parenthesis tokens to indicate grouping. Since bracket analysis may require the insertion of several tokens for a single input token, the bracket analyser places its output tokens in an internal buffer that can store several tokens. Only when this buffer is empty, does it read the next token of input from the lexical analyser.

After bracket analysis, parsing is straightforward. It uses an LALR(1) parser generated by ML-YACC. The syntax of action notation has a flat structure, where everything is a term, i.e. the syntax does not contain non-terminals for action and yielder. Precedence is handled by using postfix, prefix, and infix terms. The overall structure of the grammar is:

```
term          ::= prefix-term
                | prefix-term infix-op prefix-term | ...                (4-5)

prefix-term   ::= postfix-term
                | prefix-op postfix-term                                (4-6)

postfix-term  ::= simple-term
                | simple-term postfix-op                                (4-7)

simple-term   ::= ... | name | natural | token | "it" | "(" term ")"    (4-8)

infix-op      ::= "trap" | "with" | "is"                                (4-9)
```

```
prefix-op      ::=  "store" term "in" | "bind" term "to"
                 |  "allocate" | "a" | ...                    (4-10)

postfix-op     ::=  "[" term "]"                              (4-11)
```

Associativity is handled by allowing unbracketed lists of the *same* infix combinator. For example, the associative infix operator "then" has the following syntactic production:

```
term           ::=  ... | then-list-term | ...

then-list-term ::=  prefix-term "then" prefix-term
                 |  prefix-term "then" then-list-term         (4-12)
```

The complete syntactic specification for ACTRESS action notation is given in Appendix C.

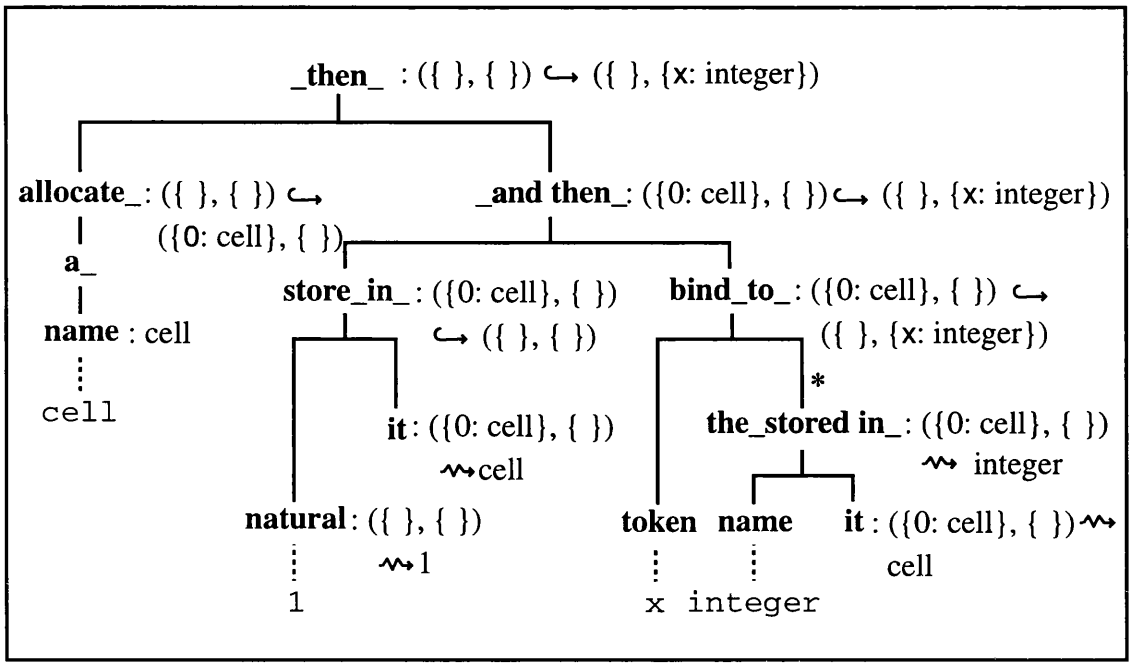
The parser constructs the corresponding action tree for the action as it is parsed. The action tree is defined using an SML `datatype` definition. Abbreviations in action notation are expanded in the action tree. For example, "give *S*" becomes "give *S* #0".

4.3 Action Notation Sort Checker

The action notation sort checker accepts an action tree and performs sort inference on it. The result is an action tree where each node of the tree has been decorated with the sort of the action tree rooted at that node. For example, Figure 4.3 shows the decorated action tree for the action given in Figure 4.2. Some of the (obvious) sorts have been omitted for space reasons. For example, the "**token**("`x`")" node has sort "token" and the "**name**("`integer`")" node has sort "integer".

Sort inference detects sort errors in the input action, such as applying an operation to an operand of the wrong sort, or using a yielder where an action is expected. Sort inference also determines the sorts of data flowing between the sub-actions in the functional and declarative facets. This information is used by the action notation code

Figure 4.3: An example of a sort-checked action


then : ({ }, { }) ↪ ({ }, {x: integer})
allocate_: ({ }, { }) ↪ ({0: cell}, { })
a_
name : cell
cell
and then: ({0: cell}, { }) ↪ ({ }, {x: integer})
store_in_: ({0: cell}, { }) ↪ ({ }, { })
it: ({0: cell}, { }) ⇝ cell
natural: ({ }, { }) ⇝ 1
1
bind_to_: ({0: cell}, { }) ↪ ({ }, {x: integer})
*
the_stored in_: ({0: cell}, { }) ⇝ integer
token
name
it: ({0: cell}, { }) ⇝ cell
x
integer

generator to guide "register allocation" in the C object code, and to determine places where code to perform a run-time sort check is needed.

The inferred sort of an action contains maps for the sorts of the transients and bindings expected by the action, and maps for the sorts of the transients and bindings produced by the action, if it completes. In general, we write the sort of an action A as:

$$A : (t, b) \hookrightarrow (t', b')$$

where t and b are the sorts of transients and bindings received by A, and where t' and b' are the sorts of transients and bindings passed out of A. For example, in Figure 4.3, the sorts given to some of the actions are:

- allocate a cell: ({ }, { }) ↪ ({0: cell}, { }). This action requires no input transients or bindings, gives a transient, labelled 0, of sort cell, and produces no bindings.

- bind "x" to the integer stored in it: ({0: cell}, { }) ↪ ({ }, {x: integer}). This action requires an input transient, labelled 0, of sort cell, requires no input bindings, gives no output transients, and produces a binding to "x" of sort integer.

Subsequently, we may use the binding for "x" with the following action:

- give the value bound to "x": ({ }, {x: integer}) $\hookrightarrow$ ({0: integer}, { }). This action requires no input transients, requires an input binding to "x" of sort integer, gives an output transient, labelled 0, of sort integer, and produces no bindings.

If an action must fail, then its sort is nothing.

Similarly, the inferred sort of a yielder contains maps for the sorts of the transients and bindings expected by the yielder, and the sort of the datum it produces (if the result is not nothing). In general, we write the sort of a yielder *Y* as:

$Y : (t, b) \rightsquigarrow S$

where *t* and *b* are the sorts of transients and bindings received by *Y*, and *S* is the sort of the datum yielded by *Y*. For example, in Figure 4.3, the sorts given to some of the yielders are:

- 1: ({ }, { }) $\rightsquigarrow$ 1. This yielder requires no input transients or bindings, and yields a datum of sort 1.

- it: ({0: cell}, { }) $\rightsquigarrow$ cell. This yielder requires an input transient, labelled 0, of sort cell, no input bindings, and yields a datum of sort cell.

- the integer stored in it: ({0: cell}, { }) $\rightsquigarrow$ integer. This yielder requires an input transient, labelled 0, of sort cell, no input bindings, and yields a datum of sort integer.

We may use a binding for “x” with the following yielder:

- the value bound to "x": ({ }, {x: integer}) $\rightsquigarrow$ integer. This yielder requires no input transients, requires an input binding to “x” of sort integer, and yields a datum of sort integer.

The notation used for the sorts of actions and yielders is explained in more detail in Chapter 5.

One of the most important tasks of the action notation sort checker is to indicate the places in the action tree where a run-time sort check is required. The action notation code generator must produce code at these points to check that the datum is of the required sort. For example, consider the point marked (*) in the action tree given in Figure 4.3. If a cell can contain a datum with a sort other than integer, then the yielder “the integer stored in it” will need to check that the actual datum fetched from storage is of the required sort (integer). If, however, a cell can only contain an integer, then no run-time sort check is required. Since run-time checks are not always required, the sort checker must determine if a check is required at each of the possible points in the action tree.

Sort inference is a complex process. It requires three passes over the action tree to perform, and uses a sophisticated unification-based algorithm to infer the sorts of data being used. Chapter 5 discusses the sorts used in ACTRESS action notation, including the sorts given to actions and yielders. Chapter 6 discusses the sort inference algorithm for action notation in detail. Finally, Chapter 7 discusses the proof of soundness for our sort inference algorithm.

4.4 Action Notation Code Generator

The translation of actions into C object code is done by the action notation code generator. An action is translated to a C statement sequence, and a yielder is translated

Figure 4.4: An example of code generation

```
 1      #include "runtime.h"
 2
 3      DATUM _d1; BINDINGS _b1;
 4
 5      int main()
 6      {
 7        _d1 = _ALLOCATE_A_CELL();
 8        *_d1.datum.cell = _MAKE_INTEGER(1);
 9        _b1 = _BIND("x", *_d1.datum.cell);
10
11        exit(0);
12      }
```

to a C expression. In the generated code, transients and bindings are passed in "registers"—C variables allocated by the code generator and declared in the object code. A register allocation discipline is necessary: the flow of data between actions must guide the allocation and deallocation of registers. The code generator is guided by the information received from the sort checker. Figure 4.4 shows the code generated for the sort-checked action in Figure 4.3 (here we have assumed that a run-time sort check is not required).

Each transient datum is contained in a special kind of register called a *d-register.* For example, the action "give *d* label #*n*" is translated to an assignment of the value of *d* to a d-register allocated at translation time (d_i). For example, line 7 in Figure 4.4 illustrates the storing of data in a d-register. The translation process must note the association between *n* and d_i. Thus "the *S* #*n*" is translated to a fetch from the d-register associated with *n*. For example, lines 8 and 9 in Figure 4.4 illustrate the code "`_d1`" generated for the yielder "it" (which is an abbreviation for "the datum #0"). In general, a run-time sort check may be necessary for "the *S* #*n*" to guarantee that the content of the register is of sort *S*; the code generator is warned by the sort checker and generates the necessary code.

At run-time, a second kind of register called a *b-register* is used to contain a set of bindings. (The set of bindings is represented by a linked-list.) The translation of "bind *k* to *d*" is just an assignment of a single binding to a b-register, $\mathtt{b}_i$, allocated at translation-time. Such a binding is built by an auxiliary function (`_BIND(`*k*`,`*d*`)`). For example, line 9 in Figure 4.4 illustrates the code generated for the action "bind "x" to the integer stored in it". The translation of "the *S* bound to *k*" is just a call to another auxiliary function (`_BOUND(`*k*`,`$\mathtt{b}_i$`)`) that looks up what datum is bound to token *k* in the register $\mathtt{b}_i$ (which is determined at translation time). Again, the code generator may have to generate the code for a run-time sort check.

Storage is represented by an array, `storage`, of datum values (which is declared as part of the run-time system). An individual cell is represented by a pointer to an element of the array. The translations of "store *d* in *c*" and "the *S* stored in *c*" are straightforward, and involve assignment via and dereferencing of the corresponding pointer denoted by the value of the cell *c*. For example, in Figure 4.4, "`*_d1.datum.cell = ...`" in line 8 illustrates the translation of a "store" action, and "`*_d1.datum.cell`" in line 9 illustrates the translation of a "stored in" yielder. Using a pointer to represent a cell gives the run-time system flexibility in the allocation mechanism used. For example, cells can also be allocated dynamically on the heap without modifying the code generator. This use of storage is exploited by the action notation transformer, which classifies the allocation of cells as static or dynamic, and allocates them from different storage areas as appropriate.

4.5 Action Notation Interpreter

The action notation interpreter accepts an action tree, performs that action, and reports its outcome. The outcome includes the transients, bindings and storage produced by the action, if it completes, or an indication of failure otherwise. For example, the outcome for the action of Figure 4.2 indicates that the action completes giving no transients, producing a binding for x to a cell *c*, and a store with cell *c* containing the value 1.

The non-deterministic action “A_1 or A_2” is interpreted as follows. A random number is generated to determine whether A_1 or A_2 should be tried first. The other sub-action is interpreted only if the first one fails. This provides some of the dynamic nature of the “or” combinator.

An abstraction is represented by a triple (A, t, b), where A is the incorporated action, t is a set of transients and b is a set of bindings. These fields are supplied by the “abstraction”, “with” and “closure” operations respectively. The “enact” action is interpreted by interpreting the action A, supplying the transients t and bindings b.

The interpreter is derived from Mosses’ operational semantics of action notation given in [Mos92]. Moreover, it implements nearly all of action notation. A full description can be found in [Mou93b].

4.6 Action Notation Transformer

The action notation transformer is a recently added module that can be used to improve the quality of the code generated for an action at the expense of extra time taken to compile it. The transformer ($transform_{\mathcal{A}}$) is applied to a decorated action tree after sort checking and produces an output decorated action tree that has been simplified. This is performed according to the transformation laws developed by Moura [Mou93a].

The action notation transformer can be used to build an improved version of the action notation compiler:

$$compile_{\mathcal{A}} \quad = \quad encode_{\mathcal{A}} \circ transform_{\mathcal{A}} \circ check_{\mathcal{A}} \circ parse_{\mathcal{A}} \qquad \textbf{(4-13)}$$

and an improved compiler for a language $\mathcal{L}$:

$$compile_{\mathcal{L}} \quad = \quad encode_{\mathcal{A}} \circ transform_{\mathcal{A}} \circ check_{\mathcal{A}} \circ act_{\mathcal{L}} \circ parse_{\mathcal{L}} \qquad \textbf{(4-14)}$$

The main area for improvement is the manipulation of bindings at run-time. For most actions, it is possible to analyse the action, and to identify the "bind" action which produces the binding used in each occurrence of "the *S* bound to *k*". If an occurrence of "the *S* bound to *k*" can be linked to a unique "bind" action, then it is possible to eliminate that occurrence of the yielder. If all occurrences of "the *S* bound to *k*" are eliminated, then the action "bind *k* to *Y*" can also be eliminated. If all uses of the declarative facet can be eliminated from an action, then substantial savings can be achieved in both the size and speed of the generated object code.

The bindings in an action can be divided into two types: bindings for known values, and bindings for unknown values. This classification is based on the sort information produced by the action notation sort checker. A known value is represented by a binding to an individual sort, for example, "x: 3". An unknown value is represented by a binding to a proper sort, for example, "x: integer". Bindings for known values can be eliminated by replacing each use of the binding by the known value. Unknown bindings are slightly harder to eliminate. They can, however, be treated in a similar manner if the unknown value is first stored in a known storage cell. The binding can now be eliminated as before, but here each use of the binding is replaced by a fetch of the unknown value from the known cell.

The action notation transformer eliminates the bindings one-by-one from the action. It is therefore possible that not all bindings will be removed from the action.

In order to eliminate all of the bindings from an action, the action must possess two properties. First, the action must be "statically-scoped", i.e. each use of "the_bound to_" must be matched to a corresponding, unique "bind" action. Second, the action must have a known space requirement, so the code generator can perform storage allocation at compile-time and generate known cell values. This restricts the use of the "allocate" action.

Figure 4.5: Input for the actioneer generator (extract)

```
(*) evaluate_ :: Expression -> action[giving a value]
      [using current bindings!current storage].

(1) evaluate [[ IDENT ~I:Identifier ]] =
      give the value bound to ~I
      or give the value stored in the cell bound to ~I .

(*) execute_ :: Statement -> action[storing!diverging]
      [using current bindings!current storage] .

(2) execute [[ SEQ ~S1:Statement ~S2:Statement ]] =
      execute ~S1
      and then execute ~S2 .

(3) execute [[ WHILE ~E:Expression ~S:Statement ]] =
      unfolding
      |evaluate ~E
      |then
      | |execute ~S and then unfold
      | |else complete .

(4) execute [[ LET ~D:Declaration ~S:Statement ]] =
      furthermore elaborate ~D
      hence execute ~S .

(*) elaborate_ :: Declaration ->
      action[storing!binding]
      [using current bindings!current storage] .

(5) elaborate [[ CONST ~I:Identifier ~E:Expression ]] =
      evaluate ~E
      then bind ~I to the value .
```

4.7 Actioneer Generator

The actioneer generator accepts an ASCII representation of the (dynamic) semantics of a programming language $\mathcal{L}$, and from it generates a simple translator from an $\mathcal{L}$-abstract syntax tree to its corresponding action tree—an *actioneer* for $\mathcal{L}$. An extract from the ASCII version of the NANO-Δ specification (given in Chapter 2) is shown in Figure 4.5. Also, the abstract syntax definition for language $\mathcal{L}$ must be directly expressed in SML. An extract from the abstract syntax for NANO-Δ is shown in

Figure 4.6: Abstract syntax for NANO-Δ (extract)

```
datatype Expression = IDENT of string | ...

and Statement = SEQ of Statement * Statement |
        WHILE of Expression * Statement |
        LET of Declaration * Statement | ...

and Declaration = CONST of String * Expression | ...
```

Figure 4.6.

The generated actioneer consists of a number of SML functions, one for each of the semantic functions in the dynamic semantics. Each function has a number of clauses, one for each of the clauses in the dynamic semantics. Each function clause maps one of the syntactic forms of the language into its corresponding piece of action tree. The function clause contains calls to the other functions to handle the sub-phrases of the construct being translated. The result is a single, large action tree representing the entire source program. The corresponding extract from the generated actioneer for NANO-Δ is shown in Figure 4.7.

The actioneer generator performs little error-checking beyond simple syntactic correctness. In particular, sort errors such as applying an action-combinator to a yielder argument are not detected until a program using the erroneous action is sort checked, i.e. at compile time rather than at compiler-generation time. Also, logical errors in the specification are not detected. For example, in Figure 4.5, if the language designer had mistakenly written "execute ~D", instead of "elaborate ~D", then the actioneer generator will not report an error, but the generated actioneer will not compile as D is of type Declaration and not Statement. This situation can be improved by using the more sophisticated actioneer generator discussed in Chapter 8.

Figure 4.7: A generated actioneer (extract)

```
(* evaluate : Expression -> Action *)

fun evaluate (IDENT I) =
      OR(GIVE(BOUND_TO(NAME("value"), TOKEN(I))
        GIVE(STORED_IN(NAME("value"),
          BOUND_TO(NAME("cell"),TOKEN(I))))))

(* execute : Statement -> Action *)

and execute (SEQ (S1,S2)) =
        AND_THEN(execute S1, execute S2)

|  execute (WHILE (E,S)) =
        UNFOLDING(THEN(evaluate E,
          ELSE(AND_THEN(execute S, UNFOLD),COMPLETE)))

|  execute (LET (D,S)) =
        HENCE(FURTHERMORE(elaborate D), execute S)

(* elaborate : Declaration -> Action *)

and elaborate (CONST (I,E)) =
        THEN (evaluate E, BIND(TOKEN(I),
          THE(NAME("value"),NATURAL("0")))
```

Chapter 5

Sorts in Action Notation

In Chapter 2, we saw that values in action semantics are classified into different sorts. In this chapter, we present a detailed discussion of these sorts. We begin by describing the sorts in action notation given in [Mos92], which we will refer to as *standard* action notation. We argue, however, that the standard notation for describing the sorts of actions and yielders is unsuitable for use in ACTRESS, and so we present our own notation for describing these sorts. We use this notation as the basis for our sort inference algorithm which is discussed in Chapter 6.

5.1 Sorts in Standard Action Notation

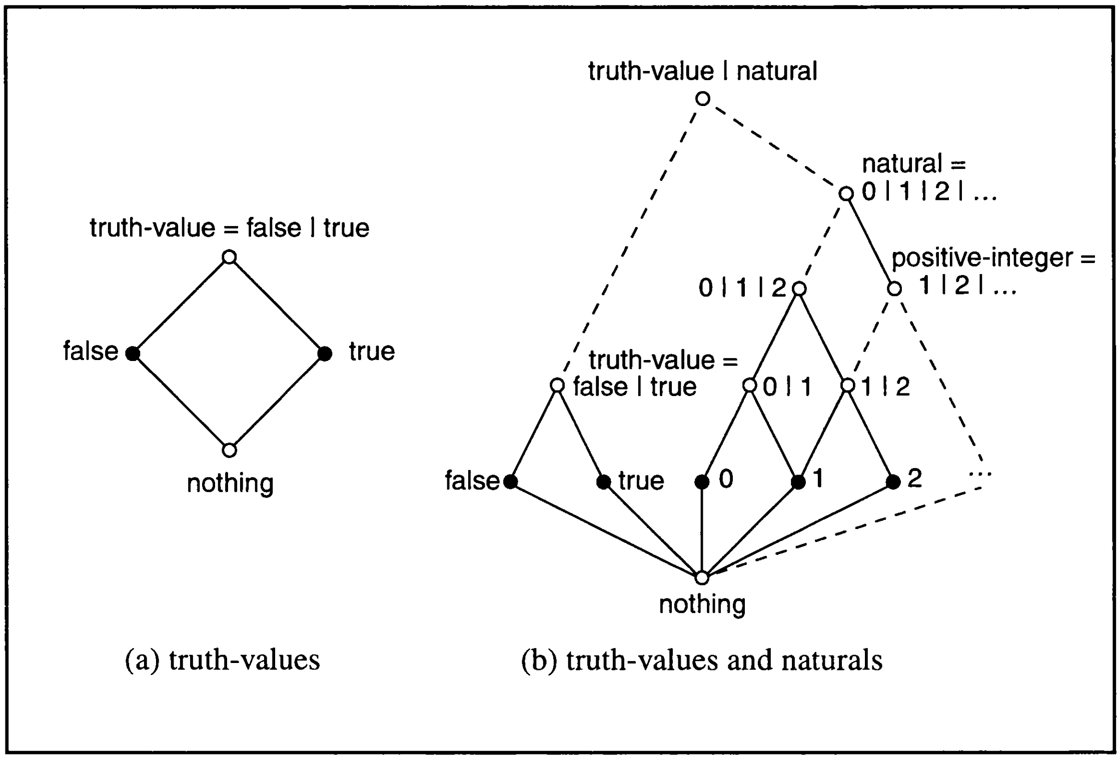
This section describes the sorts in standard action notation. It also describes the standard notation for specifying subsorts of data, actions and yielders.

5.1.1 Background

The theoretical foundation of action notation is Mosses' *unified algebras* [Mos92]. This algebraic framework elegantly solves some of the problems that beset older algebraic frameworks, by the simple (according to Mosses) expedient of abandoning the usual sharp distinction between values and sorts.

In a unified algebra, a *sort* is just a classification of *individuals*. No distinction is made between an individual and the singleton sort that classifies just that individual. A

Figure 5.1: Example sort hierarchies



sort which classifies several individuals is known as a *proper* sort. The least sort, nothing, is the classification of no individuals, and is also a proper sort. Sorts are partially ordered by a subsort relation, "≤". The *join* of two sorts (S_1 I S_2) is their least upper bound with respect to "≤", and the *meet* of two sorts (S_1 & S_2) is their greatest lower bound with respect to "≤". The sorts form a distributive lattice. The notation "I : S" asserts that individual I belongs to sort S. The subsort relation is the obvious one, namely $S_1 \leq S_2$ if and only if S_2 contains all the individuals of S_1.

For example, in Figure 5.1(a), the universe of discourse consists of the truth values. The individuals are false and true. The sorts are nothing, false, true, and false I true. In this example, nothing and truth-value = false I true are the only proper sorts, i.e., sorts that are not individuals. The nodes of the graph represent the sorts (individuals being shaded black and proper sorts white); the edges of the graph represent the "≤" relation.

In Figure 5.1(b), the universe of discourse consists of not only the truth values but also the natural numbers (individuals 0, 1, 2, ...). In this example there are infinitely many sorts, of which only a few are shown. Among the interesting proper sorts are 0 | 1 | 2, 1 | 2 | 3 | ... (also known as positive-integer), 0 | 1 | 2 | 3 | ... (also known as natural), and truth-value | natural. There are also many less useful sorts, such as 2 | true.

One benefit of unified algebras is that operations may be defined uniformly over proper sorts as well as individuals. For example, the operation "successor_" not only maps 0 to 1, 1 to 2, ...; it also maps 0 | 1 to 1 | 2, ..., and positive-integer to natural[1]. Also, *all* operations are monotone, namely:

$$S_1 \le S'_1, \ldots, S_n \le S'_n \Rightarrow O(S_1, \ldots, S_n) \le O(S'_1, \ldots, S'_n)$$

for an n-ary operation O, and sorts S_i and S'_i $(1 \le i \le n)$.

The sort lattice contains all of the action notation values: actions, yielders and data. However they are separated in the lattice into subsorts of distinct sorts denoted by action, yielder and data respectively.

5.1.2 Data Sorts

In standard data notation, the sort data classifies all data values. Standard data notation contains definitions for a variety of *basic* sorts of data including integers, truth values, characters, and strings. It also specifies most of the expected operations over these sorts, for example: successor_, sum(_,_), and either(_,_). Standard data notation also contains definitions of *constructed* sorts such as (heterogeneous) lists, maps, (syntax) trees and tuples. A complete description can be found in [Mos92], Appendix E. In fact, data represents the sort of (flat[2]) tuples whose elements are of sort datum. So all non-tuple sorts are also subsorts of datum.

A typical definition of a constructed sort defines three things:

1. Indeed, these infinite sorts are defined by the recursive equations positive-integer = successor natural; natural = 0 | positive-integer (*disjoint*).
2. i.e. there are no tuples of tuples.

- A top element for all values of that sort. For example, all lists are subsorts of the sort list.

- An operation for specifying proper subsorts of the constructed sort. For example, the operation "list[_]" maps sorts of data to sorts of lists. In particular, list[truth-value] is the sort of all lists of truth values; list[1] is the sort of all lists of ones[3]; list[natural] is the sort of all lists of natural numbers; and list[truth-value | natural] is the sort of all lists of truth values and natural numbers (a sort of heterogeneous lists).

- Some operations for building individuals of the sort. For example, "list of_" builds a list from a tuple of values[4], and "concatenation_" concatenates a tuple of lists[5]. In particular, "list of 1" is the singleton list containing the individual one, and "concatenation (the list#1, the list#2)" is the concatenation of the two lists.

For all practical purposes, we may view sorts of data as sets, nothing as the empty set, "$I : S$" as set membership, "$S_1 \leq S_2$" as set inclusion, "$S_1 \mid S_2$" as set union, and "S_1 & S_2" as set intersection.

5.1.3 Action Sorts

In standard action notation, the sort action classifies all actions. A subsort of actions is characterised either by restricting its *incomes* (the data it may use), or by restricting its *outcomes* (i.e., whether it completes, fails, or diverges, and what data it passes out if it does complete), or by restricting both its incomes and its outcomes. For example: the sort "action[using current bindings]" classifies actions that may use the bindings propagated into them; the sort "action[giving an integer]" classifies actions that each gives a datum of sort integer; the sort "action[binding]" classifies actions that may

[3] Note that here list[_] maps an individual to a sort.
[4] Or, in ACTRESS, a singleton list from a single value.
[5] Limited to concatenation (_,_) in ACTRESS.

Table 5.1: Action sorts in standard action notation

Facets/Outcomes

- outcome = giving data I binding I failing I ❑.
- giving _ :: data → outcome (*strict, linear*) .
- completing = giving ( ) .
- failing = nothing .

Facets/Incomes

- income = given data I current bindings I current storage I ❑ .
- given _ :: data → income (*strict, linear*) .
- given _#_ :: data, positive-integer → income (*strict, linear*) .

Facets/Actions

- _ [_] :: action, outcome → action .
- _ [using _] :: action, income → action .

$O_1, O_2 \leq$ outcome ; $I_1, I_2 \leq$ income ; $A, A_1, A_2 \leq$ action $\Rightarrow$

(1) A[outcome] = A ;

(2) $A[O_1][O_2] = A[O_1$ & $O_2]$;

(3) $A_1[O_1]$ & $A_2[O_2] = (A_1$ & $A_2)[O_1$ & $O_2]$;

(4) A[using income] = A ;

(5) A[using I_1][using I_2] = A[using I_1 & I_2] ;

(6) A_1[using I_1] & A_2[using I_2] = (A_1 & A_2)[using I_1 & I_2] .

produce bindings; the sort "action[storing]" classifies actions that may effect changes in storage; and the sort "action[binding][using current bindings]" classifies actions that may both use and produce bindings.

A part of the standard notation for specifying action sorts is given in Table 5.1. A full description can be found in [Mos92], Section B.9. Some examples of actions and their sorts in standard action notation are[6]:

- bind "n" to 7 : action[completing|binding][using nothing]

This action *must* require no input information, *must* produce empty transients, and *may* produce non-empty bindings. From this sort, we cannot tell that the action must produce a single binding to "n" to the datum 7.

- bind "n" to the integer #1 : action[completing|binding][using the given integer]

This action *may* require a transient of sort integer, *must* produce empty transients, and *may* produce non-empty bindings. From this sort, we cannot tell that the action must produce a single binding to "n" to the received transient.

- give sum (the integer#1, the integer#2) label #3 :

  action[giving an integer][using the given (integer,integer)]

This action *may* require a pair of transients, both of sort integer, and *must* produce a transient of sort integer.

- rebind : action[completing|binding][using current bindings]

This action *may* require non-empty bindings, and *must* produce empty transients, and *may* produce non-empty bindings. From this sort, we cannot tell that the action will produce the same set of bindings that it receives.

Some of the properties of the notation for action sorts are slightly counter-intuitive. For example:

action[binding|storing]	≠	action[binding][storing]	=	action[binding & storing]
			=	action[nothing]
			=	action[failing]

Multiple, possible outcomes can, therefore, only be specified all at once, e.g.

[6] Note, in these and later examples, the actions are written in ACTRESS action notation, but the sorts are written in standard action notation. This is because the standard notation for action sorts was introduced after the ACTRESS action notation was defined.

"action[binding I storing I giving an integer]". A similar argument also applies to incomes. Also, the standard notation for action sorts provides no means of specifying the sorts of actions *without* particular properties. For example, we cannot specify the sorts of actions which *do not* produce bindings (such as the third example above). Trying to introduce such notation leads to problems with monotonicity. For our purposes, however, knowing such behaviour would be extremely useful, for example, when sort-checking specifications.

5.1.4 Abstraction Sorts

In action notation, an abstraction incorporates an action, which is performed whenever the abstraction is enacted. It follows that abstraction sorts are isomorphic to action sorts. The same notation for restricting the incomes and outcomes of actions is used for abstractions.

Some examples of abstractions and their sorts are:

- abstraction (bind "n" to 7) : abstraction[completinglbinding][using nothing]
- abstraction (bind "n" to the integer #1) :
  abstraction[completinglbinding][using the given integer]

5.1.5 Yielder Sorts

Analogous to actions, all yielders are classified by the sort yielder. Moreover a subsort of yielders is characterised either by restricting its incomes, or by restricting the sort of the datum it yields.

A part of the standard notation for specifying yielder sorts is given in Table 5.2. A full description can be found in [Mos92], Section B.9. Again, multiple incomes must be specified all at once to give the intended sort.

Some examples of yielders and their sorts are:

Table 5.2: Yielder sorts in standard action notation

Facets/Yielders

- _ [_] :: yielder, datum $\longrightarrow$ yielder .
- _ [using _] :: yielder, income $\longrightarrow$ yielder .

$d_1, d_2 \leq$ datum ; $I_1, I_2 \leq$ income ; $Y, Y_1, Y_2 \leq$ yielder $\Rightarrow$

(1) Y [datum] = Y ;

(2) $Y[d_1][d_2] = Y[d_1$ & $d_2]$;

(3) $Y_1[d_1]$ & $Y_2[d_2] = (Y_1$ & $Y_2)[d_1$ & $d_2]$;

(4) Y [using income] = Y ;

(5) Y [using I_1][using I_2] = Y [using I_1 & I_2] ;

(6) Y_1[using I_1] & Y_2[using I_2] = (Y_1 & Y_2)[using I_1 & I_2] ;

(7) datum = datum[using nothing] .

- the integer bound to "n" : yielder[integer][using current bindings]

This yielder *may* use the current bindings, and *must* yield a datum of sort integer. From this sort, we cannot tell that the yielder only uses the binding for “n”.

- the integer#1 : yielder[integer][using the given integer]

This yielder *may* use the given transient of sort integer. and *must* yield a datum of sort integer.

- the integer stored in the cell bound to "x" :

  yielder[integer][using current bindings|current storage]

This yielder *may* use the current bindings or the current storage, and *must* yield a datum of sort integer. From this sort, we cannot tell that the yielder only uses the binding for “x”.

5.1.6 Disadvantages of the Standard Sort Notation

The standard notation provided for specifying the sorts of actions, yielders and abstractions is useful when writing specifications. For example, it is useful when specifying the sort of an action produced by applying a semantic function. Unfortunately, the notation is unsuitable for our purposes within a compiler generator—it is too imprecise. For example, we want to be able to specify the set of bindings received or produced by an action, or to specify that an action always produces empty bindings. Moreover, the algebraic properties of the standard sort notation makes it hard to combine sorts. This is necessary to calculate the sort of an action combinator given the sorts of its sub-actions. For example, consider the action "A_1 then A_2", the standard notation cannot assert that the sort of transients produced by A_1 must match the sort of transients required by A_2 (to attempt to do so violates monotonicity). For further examples of the limitations of the standard sort notation, the reader should consider the sorts for actions given in [Mos92] Section B.9.

Within the ACTRESS system, therefore, we must develop our own notation for the sorts of actions, yielders, and abstractions. We present our sort notation in the next section.

5.2 Sorts in ACTRESS Action Notation

The system of data sorts used in ACTRESS is a restricted version of the one found in action semantics. For example, in ACTRESS there are no tuple sorts, and so the sort data is not required. However, our sorts for actions, abstractions, and yielders are more expressive than those standard action notation (at least for the transients and bindings).

5.2.1 Data Sorts

In the ACTRESS system, we deal only with sorts that can be finitely expressed, for example, we cannot handle sort definitions such as "positive-integer = successor

Table 5.3: Syntax of data sorts in ACTRESS

(data sorts)	S ::= nothing | datum | I | B | $C[S]$ | S_1 | S_2 | S_1 & S_2
(basic individuals)	I ::= false | true | 0 | 1 | 2 | ...
(basic sorts)	B ::= truth-value | integer | ...
(sort constructors)	C ::= list | cell | ...

natural", since calculating this sort involves an infinite number of applications of successor. We therefore restrict data sort terms to those generated by the BNF grammar in Table 5.3.

This class of sorts has the following useful properties:

- The *basic individuals* are partitioned into a number of *basic sorts*, such that every basic individual belongs to a unique basic sort. Thus we can talk about *the* basic sort of a given basic individual. Infinite subsorts of basic sorts are not expressible. For example, the sorts natural and positive-integer are not expressible within ACTRESS.

- Individuals of constructed sorts are not expressible in this syntax. As we saw in Section 5.1.2, an individual of a constructed sort is built by applying a data operation—something that is only evaluated when the action is performed. An individual of a constructed sort could be represented if the result of applying the data operation were calculated. However, this would either require special knowledge of standard data operations, or the ability to evaluate arbitrary data terms at compile time.

- There are algorithms to compute "$I : S$", "$S_1 \leq S_2$", and "S_1 & S_2", for an arbitrary individual I and arbitrary sort terms S, S_1, S_2. This is shown in Section 5.2.1.1.

- Every sort term can be reduced to a finite canonical sort term, which is of the form $S_1 \mid \ldots \mid S_n$, where $n > 0$ and each S_i is a basic individual, a basic sort, or a sort constructor applied to a canonical sort term. In particular, "&" can always be eliminated. This is shown in Section 5.2.1.2.

5.2.1.1 Algorithms for Data Sort Operations

In this section, we consider the algorithms for computing "$I : S$", "$S_1 \leq S_2$", and "S_1 & S_2". Note, however, that "$I : S$" and "$S_1 \leq S_2$" are equivalent to "I & $S = I$" and "S_1 & $S_2 = S_1$" respectively. Therefore, we only have to give the algorithm for "S_1 & S_2". We calculate "S_1 & S_2" using the algorithm given in Figure 5.2, where S_1 & S_2 = *meet* S_1 S_2. In the algorithm, we have to distinguish between a sort term that is the join of two (or more) sorts and one which is not. We use the variable P (for primary sort) to range over sorts which are not joins.

It is straightforward to prove that "*meet* S_1 S_2" results in a sort S that does not contain any occurrences of "&". The proof is by structural induction over the syntax of sorts. Finally, we must prove that *meet* is commutative. The proof of commutativity is given in Section D.1.

5.2.1.2 Normalisation of Data Sorts

In this section, we consider the normalisation of data sorts. We require that an arbitrary data sort S can always be represented in the form $S_1 \mid \ldots \mid S_n$, where $n > 0$ and each S_i is either a basic individual or a basic sort or a sort constructor applied to a canonical sort term. The algorithm to convert a sort S to normal form is given in Figure 5.3.

Again, it is straightforward to prove that "*normalise* S" results in a sort of the form $S_1 \mid \ldots \mid S_n$, where $n > 0$, and in particular, none of the S_i contain occurrences of "&". The proof that "*normalise* S" does indeed return a sort in normal form is given in Section D.2.

Figure 5.2: Algorithm for *meet*

```
meet :: data-sort → data-sort → data-sort

meet  nothing   $S_2$  =  nothing

meet  datum     $S_2$  =  case $S_2$ of
        $I_2$                ⇒ $I_2$
        $B_2$                ⇒ $B_2$
        nothing             ⇒ nothing
        datum               ⇒ datum
        $C_2[S'_2]$          ⇒ $C_2$[meet datum $S'_2$]
        $S_{2a}$ & $S_{2b}$   ⇒ meet $S_{2a}$ $S_{2b}$
        $P_2$ | $S'_2$        ⇒ (meet datum $P_2$) | (meet datum $S'_2$)

meet  $I_1$     $S_2$  = case $S_2$ of
        $I_2$                ⇒ if $I_1 = I_2$ then $I_1$ else nothing
        $B_2$                ⇒ if $I_1 \in B_2$ then $I_1$ else nothing
        nothing             ⇒ nothing
        datum               ⇒ $I_1$
        $C[S'_2]$            ⇒ nothing
        $S_{2a}$ & $S_{2b}$   ⇒ let $S'_2$ = meet $S_{2a}$ $S_{2b}$ in meet $I_1$ $S'_2$
        $P_2$ | $S'_2$        ⇒ (meet $I_1$ $P_2$) | (meet $I_1$ $S'_2$)

meet  $B_1$     $S_2$  = case $S_2$ of
        $I_2$                ⇒ if $I_2 \in B_1$ then $I_2$ else nothing
        $B_2$                ⇒ if $B_1 = B_2$ then $B_1$ else nothing
        nothing             ⇒ nothing
        datum               ⇒ $B_1$
        $C[S'_2]$            ⇒ nothing
        $S_{2a}$ & $S_{2b}$   ⇒ let $S'_2$ = meet $S_{2a}$ $S_{2b}$ in meet $B_1$ $S'_2$
        $P_2$ | $S'_2$        ⇒ (meet $B_1$ $P_2$) | (meet $B_1$ $S'_2$)

meet  $C_1[S'_1]$  $S_2$  = case $S_2$ of
        $I_2$                ⇒ nothing
        $B_2$                ⇒ nothing
        nothing             ⇒ nothing
        datum               ⇒ $C_1$[meet datum $S'_1$]
        $C_2[S'_2]$          ⇒ if $C_1 = C_2$ then $C_1$[meet $S'_1$ $S'_2$] else nothing
        $S_{2a}$ & $S_{2b}$   ⇒ let $S'_2$ = meet $S_{2a}$ $S_{2b}$ in meet $C_1[S'_1]$ $S'_2$
        $P_2$ | $S'_2$        ⇒ (meet $C_1[S'_1]$ $P_2$) | (meet $C_1[S'_1]$ $S'_2$)

meet  ($P_1$ | $S'_1$)  $S_2$  = case $S_2$ of
        $S_{2a}$ & $S_{2b}$   ⇒ let $S'_2$ = meet $S_{2a}$ $S_{2b}$ in (meet $P_1$ $S'_2$) | (meet $S'_1$ $S'_2$)
        _                   ⇒ (meet $P_1$ $S_2$) | (meet $S'_1$ $S_2$)

meet  ($S_{1a}$ & $S_{1b}$) $S_2$  =  let $S'_1$ = meet $S_{1a}$ $S_{1b}$ in meet $S'_1$ $S_2$
```

Figure 5.3: Algorithm for *normalise*

```
normalise :: data-sort → data-sort

normalise   nothing       = nothing
normalise   datum         = datum
normalise   $I$           = $I$
normalise   $B$           = $B$
normalise   $C[S]$        = let $S'$ = normalise $S$ in $C[S']$
normalise   $(S_1 \& S_2)$  = let $S'$ = meet $S_1$ $S_2$ in normalise $S'$

normalise   $(P_1 \mid S_2)$  = let $P_1'$ = normalise $P_1$
                          $S_2'$ = normalise $S_2$
                      in
                          prune $(P_1' \mid S_2')$

where prune $(P_1 \mid \ldots \mid P_n)$ = if $\exists\, i, j$ s.t. $i \neq j$ and subsort $P_i$ $P_j$
                              then   prune $(P_1 \mid \ldots \mid P_{i-1} \mid P_{i+1} \mid \ldots \mid P_n)$
                              else   $(P_1 \mid \ldots \mid P_n)$
```

5.2.2 Action Sorts

In ACTRESS, we are not concerned with all possible classifications of actions. For example, we are not concerned with whether an action may diverge or not[7]. Furthermore, we are only interested in the sort information that can be inferred without performing the action. We cannot concern ourselves with the imperative facet, as storage typically relies on the dynamic allocation of cells. We can consider the functional and declarative facets, since, for a particular action, we will know the domains of the transients and bindings used in the action.

The sort of a set of bindings may be represented by a *record sort.* For example, the record sort {x: integer, y: truth-value}, represents the fact that x is bound to an unknown datum of sort integer and y is bound to an unknown datum of sort truth-value. Other examples of record sorts are {x: integer, y: true}, where in this case y is known to be bound to true, and {x: 6, y: true}, where in this case both x and y are bound to

[7]. This is, of course, undecidable.

known data. This notation is legitimate, because the individuals 6 and true are themselves sorts. It is also convenient, because the sort of a set of bindings informs us concisely which identifiers are bound to known data (those whose sorts are individuals) and which are bound to unknown data (those whose sorts are proper sorts).

These record sorts are similar to the record types studied by Wand, Cardelli, Mitchell and others [CM89,Wan87,Wan89]. The domain of each record sort must be known, i.e., there must be no variables ranging over the domain of a record sort.

We can use record sorts to represent the sorts of bindings of a particular action, since the domain of each set of bindings (a set of identifiers) will be known statically. For any action, the set of tokens that may be used in the action can only come from the set of literal tokens *k* appearing in yielders of the form "the *S* bound to *k*". Even if the action is *dynamically scoped*, i.e. an abstraction may be closed with different sets of bindings, it can only use a binding if the token appears in the action. Moreover, the set of tokens that may be produced by an action can only come from the set of literal tokens *k* appearing in sub-actions of the form "bind *k* to *Y*". Even if parts of the action are performed several times, the set of tokens used or produced is unchanged. For example, repeatedly binding a value does not introduce new tokens. Both the set of tokens used and the set of tokens produced by the action are finite.

Similarly, we can use record sorts to represent the sorts of transients, since the domain of each set of transients (a set of labels) will also be known statically. The set of labels that may be used in the action can only come from the set of literal labels *n* appearing in yielders of the form "the *S* # *n*". The set of labels that may be produced by an action can only come from the set of literal labels *n* appearing in sub-actions of the form "give *Y* label #*n*". Again, repeatedly performing parts of the action does not introduce new labels. Both the set of labels used and the set of labels produced by the action are finite.

We cannot, however, use record sorts to represent the sorts of stores, since the domain of a store (a set of cells) will be determined only dynamically. In this case, the set of cells cannot be extracted from the action, and individual cell values are only known when the action is performed. For "the *S* stored in *c*" and "store *Y* in *c*", the cell *c* is the result of evaluating a yielder, and so cannot be determined statically. Moreover, if parts of the action are performed several times, then each performance may introduce new cell values. For example, consider an allocate action that occurs inside an unfolding action.

We write the sort of an action *A* as follows:

$$A : (t, b) \hookrightarrow (t', b')$$

where *t* and *b* are the record sorts of transients and bindings received by *A*, and where *t′* and *b′* are the record sorts of transients and bindings passed out of *A* (assuming that *A* completes). If an action is ill-sorted, we write "*A* : nothing". Some examples of actions and *one* of their many possible sorts are:

- bind "n" to 7 : ({ }, { }) $\hookrightarrow$ ({ }, {n: 7})

This action receives empty transients and bindings, and it produces empty transients and a single binding of "n" to the datum 7.

- bind "n" to the integer #1 :

  ({1: integer}, {m: truth-value}) $\hookrightarrow$ ({ }, {n: integer})

This action receives a transient of sort integer, receives a binding for "m" (which it ignores), and it produces empty transients and a single binding of "n" to a datum of sort integer.

- give sum (the integer#1, the integer#2) label #3 :

  ({1: integer, 2: integer}, { }) $\hookrightarrow$ ({3: integer}, { })

This action receives two transients of sort integer and empty bindings, and it produces a transient of sort integer and empty bindings.

- rebind : ({1: integer}, {x: integer, y: truth-value}) $\hookrightarrow$
  ({ }, {x: integer, y: truth-value})

This action receives a transient of sort integer (which it ignores) and bindings for "x" and "y", and it produces empty transients and bindings for "x" and "y".

We can compare our notation for the sorts of actions with the standard notation. For example, the standard sorts for the above example actions were given in Section 5.1.3, and a comparison of these sorts with our sorts is given in Table 5.4.

Table 5.4: A comparison of standard and ACTRESS action sorts

Action	Standard Sort	ACTRESS Sort
bind "n" to 7	action[completing|binding] [using nothing]	({ }, { }) $\hookrightarrow$ ({ }, {n: 7})
bind "n" to the integer #1	action[completing|binding] [using the given integer]	({1: integer}, {m: truth-value}) $\hookrightarrow$ ({ }, {n: integer})
give sum (the integer#1, the integer#2) label #3	action[giving an integer] [using the given (integer,integer)]	({1: integer, 2: integer}, { }) $\hookrightarrow$ ({3: integer}, { })
rebind	action[completing|binding] [using current bindings]	({1: integer}, {x: integer, y: truth-value}) $\hookrightarrow$ ({ }, {x: integer, y: truth-value})

From these examples, it is clear that some of the limitations of the standard notation for action sorts have been overcome. For example, in the first sort above, we know that the action produces a single binding for "n" rather than some unknown set of bindings.

5.2.3 Abstraction Sorts

Since abstraction sorts are isomorphic to action sorts, we use similar notation for abstractions. We write the sort of an abstraction A as follows:

A : abstraction $(t, b) \hookrightarrow (t', b')$

where again t and b are the record sorts of the transients and bindings expected by the encapsulated action, and t' and b' are the record sorts of the transients and bindings passed out of the encapsulated action if it completes. Also, as abstractions are classified as data, we must augment the data sorts of Table 5.3 with abstraction sorts:

(data sorts) S ::= ... | abstraction $(t, b) \hookrightarrow (t', b')$

Some examples of abstractions and one of their many possible sorts are:

- abstraction (bind "n" to 7) :abstraction $(\{\ \}, \{\ \}) \hookrightarrow (\{\ \}, \{$n: 7$\})$
- abstraction (bind "n" to the integer #1) :

  abstraction $(\{$1: integer$\}, \{$m: truth-value$\}) \hookrightarrow (\{\ \}, \{$n: integer$\})$

5.2.4 Yielder Sorts

A yielder receives transients and bindings and yields a datum of a particular sort. The sort of a yielder Y is therefore written as follows:

$Y : (t, b) \rightsquigarrow S$

where t and b are the record sorts of transients and bindings received by Y, and S is the sort of the datum yielded by Y. If a yielder is ill-sorted, we write "Y : nothing". Some examples of yielders and one of their many possible sorts are:

- the integer bound to "n" : $(\{\}, \{$n: integer$\}) \rightsquigarrow$ integer

This yielder receives empty transients and a binding for "n" to a datum of sort integer, and yields a datum of sort integer.

- the truth-value#1 : $(\{$1: true$\}, \{\}) \rightsquigarrow$ true

This yielder receives a transient of sort true and empty bindings, and yields a datum of sort true.

- the integer stored in the cell bound to "x" : ({}, {x: cell[integer]}) $\rightsquigarrow$ integer

This yielder receives empty transients and a binding for "x" to a datum of sort cell[integer], and yields a datum of sort integer.

Comparing these sorts with those in the examples of Section 5.1.5, it is clear that some of the limitations of the standard notation for yielder sorts have been overcome.

We can improve the sort information further—we adopt the normal approach of extending types to types schemes, so our sorts become sort schemes.

5.2.5 Extending Sorts to Sort Schemes

An action, abstraction, or yielder has many sorts, as it may receive transients and bindings which it simply ignores, or it may be performed with different sets of input transients and bindings. Also, we may widen the sort of a particular datum without invalidating the sort, e.g. by replacing an individual by its corresponding basic sort, or even by datum. For example, some of the sorts of the action "bind "n" to 7" are:

({ }, { }) $\hookrightarrow$ ({ }, {n: 7})

({ }, { }) $\hookrightarrow$ ({ }, {n: integer})

({ }, { }) $\hookrightarrow$ ({ }, {n: datum})

({1: integer}, { }) $\hookrightarrow$ ({ }, {n: 7})

since any transients or bindings received by this action are ignored. These sorts are ordered by a subsort relation where "({ }, { }) $\hookrightarrow$ ({ }, {n: 7})" is the least sort in this example. Note that not all actions have a unique least sort in this framework.

We want to be able to describe the family of sorts which are valid sorts of an action. In particular, we want to identify the transients and bindings *required* by an action, i.e. the ones it actually uses or propagates. We can achieve this if we extend our

Table 5.5: Syntax of data sorts in ACTRESS

(data sort schemes)	σ	::=	nothing | datum | I | B | $C[\sigma]$ | $\sigma_1 \mid \sigma_2$ |
			σ_1 & σ_2 | θ | abstraction $(\tau, \beta) \hookrightarrow (\tau', \beta')$
(action sort schemes)	α	::=	$(\tau, \beta) \hookrightarrow (\tau', \beta')$ | nothing
(yielder sort schemes)	υ	::=	$(\tau, \beta) \rightsquigarrow \sigma$ | nothing
(transient sort schemes)	τ	::=	$\{l_1: \phi_1, \ldots, l_m: \phi_m\}[\rho \mid \gamma]$
	τ'	::=	$\{l_1: \phi_1, \ldots, l_m: \phi_m\}[\rho]$
(binding sort schemes)	β	::=	$\{k_1: \phi_1, \ldots, k_n: \phi_n\}[\rho \mid \gamma]$
	β'	::=	$\{k_1: \phi_1, \ldots, k_n: \phi_n\}[\rho]$
(field schemes)	ϕ	::=	σ | **absent** | Δ

sorts to *sort schemes*. In particular, we must extend our record sorts into *record sort schemes*. Again this closely matches the extension of record types into record type schemes in the literature.

More precisely, we will use the sort schemes generated by the grammar in Table 5.5. Labels are natural numbers and are denoted by l_i. Tokens are strings and are denoted by k_i.

The data sorts of Table 5.3 have been extended to data sort schemes. Here θ is a *sort variable* representing an unknown sort. It can be *instantiated* for a particular sort to produce different sorts of data. The definitions of basic individuals, basic sorts, and sort constructors are unchanged.

In the literature, record types are extended to record type schemes in part by allowing a suffix *row variable*. A row variable can be instantiated to a record type whose domain is disjoint from the fields explicitly stated in the record type scheme. In our record sort schemes, however, we use two different kinds of row variables. Variables denoted by ρ_i are used to denote unknown input transients or bindings that are either used or *propagated* by an action. Variables denoted by γ_i are used to denote input transients or bindings that an action simply ignores. Since no action is obliged to use all the transients or bindings passed into it, most actions have γ-variables affixed to the τ and β parts of their sort schemes. As abstraction sort schemes are isomorphic to action sort schemes, the same argument also applies. Yielder sort schemes typically only need γ-variables affixed to their (input) sort schemes to denote any unused transients or bindings that a yielder receives, as yielders do not propagate records.

The sort schemes corresponding to the example action sorts in Section 5.2.2 are:

- bind "n" to 7 : $(\{\}\gamma_1, \{\}\gamma_2) \hookrightarrow (\{\}, \{\text{n: 7}\})$
- bind "n" to the integer #1 :$(\{\text{1: integer}\}\gamma_3, \{\}\gamma_4) \hookrightarrow (\{\}, \{\text{n: integer}\})$
- give sum (the integer#1, the integer#2) label #3 :

  $(\{\text{1: integer, 2: integer}\}\gamma_5, \{\}\gamma_6) \hookrightarrow (\{\text{3: integer}\}, \{\})$
- rebind : $(\{\}\gamma_7, \{\}\rho_1) \hookrightarrow (\{\}, \{\}\rho_1)$
- furthermore bind "x" to the integer bound to "y" :

  $(\{\}\gamma_8, \{\text{y: integer}\}\rho_2) \hookrightarrow (\{\}, \{\text{x: integer, y: integer}\}\rho_2)$

The action "rebind" (which propagates the bindings it receives) is the simplest example of an action whose sort scheme contains a ρ-variable. This action is *polymorphic*, i.e., it operates uniformly over any sort of received bindings, and its sort contains a ρ-variable to reflect this polymorphism. Actions derived from "rebind", such as "furthermore *A*", are also polymorphic.

When a record sort has a row variable affixed to it, the row variable may be instantiated to any record sort with a disjoint domain. For example, consider the following action sort:

give the integer bound to "v" : $(\{\ \}\gamma_1, \{\text{v: integer}\}\gamma_2) \hookrightarrow (\{\text{0: integer}\}, \{\ \})$

In {v: integer}γ_2, the row variable γ_2 could (for example) be instantiated to {x: integer, y: truth-value}, representing the possibility that the action may receive (but ignore) bindings for x and y. Thus we have:

give the integer bound to "v" : ({ }γ_1, {v: integer, x: integer, y: truth-value})
$\hookrightarrow$ ({0: integer}, { })

There are, of course, many other possibilities. However, {v: truth-value} is *not* a possibility, because of the duplicate v field. In { }γ_1, the row variable γ_1 could be instantiated to *any* record sort, representing the possibility that the above action may receive *any* transients (but ignores them).

The sort schemes corresponding to the yielder sorts in Section 5.2.4 are:

- the integer bound to "n" : ({ }γ_1, {n: integer}γ_2) $\rightsquigarrow$ integer
- the truth-value#1 : ({1: true}γ_3, { }γ_4) $\rightsquigarrow$ true
- the integer stored in the cell bound to "x" :
  ({ }γ_5, {x: cell[integer]}γ_6) $\rightsquigarrow$ integer

We use these sorts schemes for actions, yielders and abstractions in the definition of a sort inference algorithm for ACTRESS action notation. This algorithm is discussed in the next chapter.

5.2.6 Minimal Sorts and Principal Sort Schemes

With any sort (or type) system, there are two concepts that are of interest, namely those of minimal sort and principal sort scheme.

The minimal sort of a term is defined as follows: if for a term t, we have t: S, then the sort S is *minimal* if, for all other sorts S' such that t: S', $S \leq S'$. In traditional type systems, we expect a term to have a unique minimal type.

The principal (or most general) sort scheme of a term is defined as follows: if for term *t*, we have *t*: σ, then the sort scheme σ is *principal* if, for all other sort schemes σ′ such that *t*: σ′, σ′ is an instance of σ. In traditional type inference systems, we expect to infer the principal type of a term.

Unfortunately, in our system, we have difficulties with both minimal sorts and principal sort schemes. The following examples illustrate the problems.

Consider the action "give the integer #1", we might assign the following sort to this action:

- give the integer#1 : ({1: integer}, { }) $\hookrightarrow$ ({1: integer}, { })

but we could also assign the following sort to this action:

- give the integer#1 : ({1: 123}, { }) $\hookrightarrow$ ({1: 123}, { })

Clearly, this sort can be viewed as a subsort of the first, since 123 ≤ integer. However, by the same argument, we can also assign the following sort to this action:

- give the integer#1 : ({1: 456}, { }) $\hookrightarrow$ ({1: 456}, { })

Now, this too is a subsort of the original sort, but this sort is not a subsort of "({1: 123}, { }) $\hookrightarrow$ ({1: 123}, { })" and vice versa. Also note that there is no other (valid) sort which is a subsort of both of these sorts. Hence this action has no minimal sort. The lack of a minimal sort arises in our system because individuals are allowed as sorts.

This problem is not unexpected when considering type systems with even simple sub-typing. For example, Schmidt[Sch94, page 124] describes a very simple type system for arithmetic expressions which loses the minimal typing property when the disjoint sub-types *nonnegative* and *nonpositive* of the type *integer* are introduced.

For the problem of principal sort scheme, consider the following actions and their sorts (recall that any sort is trivially a sort scheme as well):

- A_1 : ({1: integer, 2: truth-value}, { }) $\hookrightarrow$ ({ }, { })
- A_2 : ({1: truth-value, 2: integer}, { }) $\hookrightarrow$ ({ }, { })

Now consider the action "A_1 or A_2". This would be assigned the sort:

- A_1 or A_2 : ({1: integer | truth-value, 2: integer | truth-value}, { }) $\hookrightarrow$ ({ }, { })

This is the best sort we can assign to this action given our sort notation. This sort, however, suggests that the input transients {1: true, 2: false} would be acceptable to the action "A_1 or A_2", but neither A_1 nor A_2 can accept these input transients, as they do not correspond to the sorts of transients expected by these actions. So we have an action where the very best sort we can infer does not describe the precise set of valid inputs to the action.

This means that the concept of principal sort is not useful in our system, since there are occasions when the only sort scheme that could be assigned to an action also allows us to provide inputs which do not correspond with the expected inputs of the sub-actions. A principal sort scheme would only be useful if it was guaranteed to allow only well-formed inputs to an action.

The remainder of this thesis concerns itself with the soundness of a sort inference algorithm which, although incomplete, proves useful in practice.

Chapter 6

Sort Inference in Action Notation

6.1 Introduction

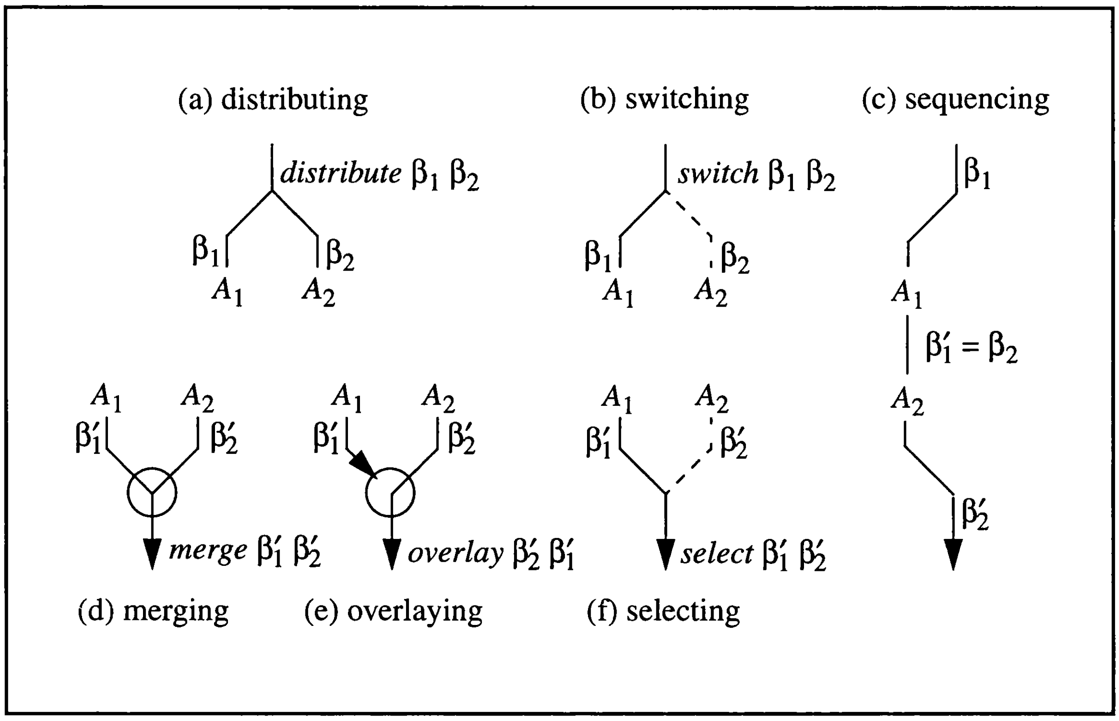
In this chapter, we are concerned with determining the sort of an action. Since an action does not explicitly state the information it requires and produces, in general, we must infer its sort by analysing the action itself. To do this, we have developed a sort inference algorithm, which can be used to determine the sort of an action by combining the sorts of the primitive actions and yielders it contains. The algorithm is presented in Section 6.3 as a set of sort inference rules. Section 6.4 gives an example of sort inference. Finally, in Section 6.5, we are concerned with the implementation of the sort inference algorithm to produce the action notation sort checker, a key part of the ACTRESS system.

We begin, however, by considering the operations required to manipulate record sort schemes in ways that are consistent with the behaviours of the different action combinators. These operations are the key to the sort inference algorithm. They allow us to write concise sort inference rules for action notation.

6.2 Auxiliary Operations

As we saw in Figure 2.1, there are several different ways that the action combinators propagate received information and combine produced information. The sort inference

Figure 6.1: Data flows in action notation and the auxiliary operations



algorithm, therefore, requires auxiliary operations that combine transient or binding record sort schemes in a way that is consistent with each of the data flows. The different data flows and their corresponding auxiliary operations are shown in Figure 6.1. The action combinators that use the various auxiliary operations are summarised in Table 6.1.

Table 6.1: Auxiliary operations

Operation name		Usage
input	*distribute*	most combinators
	switch	or, else
output	*merge*	most combinators
	select	or, else
	overlay	moreover, furthermore, before

The operations are divided into two groups: those applied to input record sort schemes, and those applied to output record sort schemes. All of the operations take two record sort schemes and produce a record sort scheme, and all except *overlay* are used with both transient and binding sort schemes. Moreover, an auxiliary operation applied to two record sort schemes will normally result in the instantiation of some of the variables of the argument sort schemes. Formally, each auxiliary operation is defined to return both a record sort scheme and a substitution. In the sort inference algorithm, however, only a single global substitution is required, and so we have chosen to leave its construction implicit, to avoid obscuring the sort inference rules themselves. So an auxiliary operation (e.g. *distribute*), which maps a pair of record sort schemes to a result record sort scheme, is actually an imperative version of the (pure) algorithm given in Section 6.2.5 (e.g. $distribute_p$).

The idea for most of the operations (*distribute*, *merge*, and *overlay*) comes from Even and Schmidt's algorithm[ES90] where they use the corresponding operations *unify-record*, S_{merge} and S_{concat} respectively. In our framework, however, they have all been enhanced to use our record sort schemes rather than those used by Even and Schmidt. Also, since Even and Schmidt's notation does not contain the "or" combinator, the *switch* and *select* operations are new.

The following four sections informally introduce each of the auxiliary operations. We present algorithms to calculate the auxiliary operations in Section 6.2.5 and consider their algebraic properties in Section 6.2.6.

6.2.1 Distribute

The auxiliary operation *distribute* combines two record sort schemes, taking the pairwise meet of any sort schemes associated with the same fields. It is used with both transient and binding schemes to combine the input sorts of the two sub-actions of a binary action combinator where both sub-actions are performed and the inputs are distributed, e.g., "and", "and then", or "moreover". Some examples of the use of *distribute* with two record sort schemes are:

- *distribute* {w: σ_1, x: σ_2} {x: σ_3, y: σ_4} = {w: σ_1, x: (σ_2 & σ_3), y: σ_4}

- *distribute* {w: σ_1, x: σ_2}ρ_1 {x: σ_3, y: σ_4} = {w: σ_1, x: (σ_2 & σ_3), y: σ_4}, where ρ_1 is instantiated to {y: σ_4}, i.e. the first argument is updated to include a field "y: σ_4" and no others.

- *distribute* {w: σ_1, x: σ_2}ρ_1 {x: σ_3, y: σ_4}ρ_2 = {w: σ_1, x: (σ_2 & σ_3), y: σ_4}ρ_3, where ρ_1 is instantiated to {y: σ_4}ρ_3, and ρ_2 is instantiated to {w: σ_1}ρ_3, i.e. the first argument is updated to include a field "y: σ_4", the second argument is updated to include a field "w: σ_1", and both arguments may include additional fields (represented by the row variable ρ_3).

In each of these examples, if the sort scheme (σ_2 & σ_3) is nothing, then the *distribute* operation returns *failure*.

The *distribute* of a record sort scheme containing a γ-variable is handled similarly to one containing a ρ-variable. The only case that requires special attention is when a record sort scheme containing a γ-variable is united with one containing a ρ-variable. Here the ρ-variable takes priority, and the resulting scheme also has a ρ-variable affixed to it. For example:

- *distribute* {w: σ_1, x: σ_2}γ_1 {y: σ_3}ρ_2 = {w: σ_1, x: σ_2, y: σ_3}ρ_3, where γ_1 is instantiated to {y: σ_3}ρ_3, and ρ_2 is instantiated to {w: σ_1}ρ_3.

Intuitively, if one sub-action propagates the received information (ρ) but the other sub-action ignores it (γ), then the whole action also propagates that information.

6.2.2 Merge

The auxiliary operation *merge* concatenates two record sort schemes, insisting that their domains are disjoint. It is used with both transient and binding schemes to combine the output sorts of the two sub-actions of a binary action combinator, where the outputs are merged and must not overlap, e.g., "and" or "and then". For example:

- *merge* {w: σ_1} {x: σ_2, y: σ_3} = {w: σ_1, x: σ_2, y: σ_3}

- *merge* {w: σ_1}ρ_1 {x: σ_2, y: σ_3} = {w: σ_1, x: σ_2, y: σ_3}ρ_2, where ρ_1 is instantiated to {x: **absent**, y: **absent**}ρ_2, i.e. the first argument is updated to exclude the "x" and "y" fields present in the second argument.

Here we are using the **absent** notation for field schemes for the first time. Remember that {x: **absent**} represents the family of record sorts in which there are *no* x-fields.

If the domains are not disjoint, then the *merge* operation returns *failure.*

6.2.3 Switch and Select

The auxiliary operations *switch* and *select* are peculiar to "or" (and its derivative "else"), and reflect the fact that this combinator performs only one of its sub-actions.

The *switch* operation is similar to the *distribute* operation, except that it takes the pairwise join of any sort schemes associated with the same fields. For example:

- *switch* {w: σ_1, x: σ_2,} {x: σ_3, y: σ_4, z: σ_5} = {w: σ_1, x: (σ_2 | σ_3), y: σ_4, z: σ_5}

The *select* operation also forms the pairwise join of the sort schemes, but it also insists on the domains of the record sort schemes being identical. Its use with "or" enforces a deliberate restriction in our sort inference algorithm, namely that the transients and bindings passed out of the two sub-actions of "or" must have identical domains—we forbid *conditional* transients and bindings. For example:

- *select* {x: σ_1, y: σ_2} {x: σ_3, y: σ_4} = {x: (σ_1 | σ_3), y: (σ_2 | σ_4)}

If the domains are not identical, then the *select* operation returns *failure.*

6.2.4 Overlay

The *overlay* operation is used in declarative action combinators where one set of bindings takes priority over another, e.g., “moreover”, “furthermore”, and “before”. It calculates the record sort scheme obtained by overlaying the second record sort scheme by the first. For example:

- *overlay* {w: σ_1, x: σ_2} {x: σ_3, y: σ_4, z: σ_5} = {w: σ_1, x: σ_2, y: σ_4, z: σ_5}
- *overlay* {w: σ_1, x: σ_2} {x: σ_3, y: σ_4, z: σ_5}ρ_1 = {w: σ_1, x: σ_2, y: σ_4, z: σ_5}ρ_2, where ρ_1 is instantiated to {w: σ_1}ρ_2, i.e. the second argument is updated to include the “w” field present in the first argument.

In both of these examples, the binding “x: σ_2” supersedes the binding “x: σ_3”.

Finally, it is possible for the two record schemes to share the same row variable. However we have the result that two record schemes with the same row variable must include the same set of fields. For example:

- *overlay* {w: σ_1, x: σ_2}ρ_1 {w: σ_3, x: σ_4}ρ_1 = {w: σ_1, x: σ_2}ρ_1.

Unfortunately there is a problem if the first record sort scheme has a row variable and the second scheme does not, or if the two record sort schemes have different row variables. For example:

- *overlay* {w: σ_1, x: σ_2}ρ_1 {x: σ_3, y: σ_4, z: σ_5} = *failure*
- *overlay* {w: σ_1, x: σ_2}ρ_1 {x: σ_3, y: σ_4, z: σ_5}ρ_2 = *failure*

In the first example, we do not know the instantiation of ρ_1, and so it is impossible to predict whether or not the “y” and “z” fields will be overlaid. For example, if ρ_1 is instantiated to {y: σ_6}, then the resulting scheme should contain the binding “y: σ_6”. However, if ρ_1 is instantiated to { }, then the resulting scheme should contain the binding “y: σ_4”. In the second example, we do not know the instantiation of ρ_2, so we cannot even predict the fields that may be overlaid. We cannot write a record sort scheme that captures this behaviour, and so these cases must be forbidden.

This restricts the class of actions that can be sort-checked. In practice this is not a problem since the cases that arise are: (i) the first record sort scheme has no row variable (corresponding to a particular set of bindings produced), and (ii) both record sort schemes share the same row variable (corresponding to harmlessly overlaying bindings with themselves). If any other case does arise, the *overlay* operation will return *failure*, and the action will be ill-sorted, even if it would complete when performed.

In our system, however, certain pathological cases have been excluded by preventing the direct use of the "rebind" action. The sort assigned to "rebind" would be $(\{\ \}\gamma, \{\ \}\rho) \hookrightarrow (\{\ \}, \{\ \}\rho)$, indicating its behaviour of merely propagating the received bindings. However, an action such as "(bind "x" to 1) moreover rebind" would involve calculating *overlay* $\{\ \}\rho$ $\{\text{x: } 1\}$, which is *failure*, and so the action is ill-sorted. Of course, this action can be performed, and it does complete.

6.2.5 Algorithms for the Auxiliary Operations

In this section, we present the algorithms used for each of the auxiliary operations. The similarities between the auxiliary operations means that each algorithm can be factorised into two components: (i) the *field operation* used to combine pairs of fields with the same name, and (ii) the *row operation* used to combine any row variables associated with the record sort schemes. This gives us the algorithms in Figure 6.2. Notice that there is a different field operation for each of the auxiliary operations, but there are only three different row operations.

The main part of the algorithm is shown in Figure 6.3 and is performed by the higher-order function *combine* which is parameterised with respect to a field operation and a row operation. The function *combine* first extends each of the input record sort schemes, r_1 and r_2, to create two new schemes with identical domains, r_1' and r_2'. This makes the pairwise application of the field operation op_f easier. The function *apply-fields* is used to do the application of the field operation op_f to each pair in turn.

Figure 6.2: Algorithms for the auxiliary operations

$$
\begin{aligned}
&\mathit{distribute}_p &&:: \text{record-scheme} \rightarrow \text{record-scheme} \rightarrow \text{substitution} \times \text{record-scheme}\\
&\mathit{merge}_p &&:: \text{record-scheme} \rightarrow \text{record-scheme} \rightarrow \text{substitution} \times \text{record-scheme}\\
&\mathit{switch}_p &&:: \text{record-scheme} \rightarrow \text{record-scheme} \rightarrow \text{substitution} \times \text{record-scheme}\\
&\mathit{select}_p &&:: \text{record-scheme} \rightarrow \text{record-scheme} \rightarrow \text{substitution} \times \text{record-scheme}\\
&\mathit{overlay}_p &&:: \text{record-scheme} \rightarrow \text{record-scheme} \rightarrow \text{substitution} \times \text{record-scheme}\\
\\
&\mathit{distribute}_p &&= \mathit{combine}\ \mathit{distribute}_{\mathit{field}}\ \mathit{distribute}_{\mathit{row}}\\
&\mathit{merge}_p &&= \mathit{combine}\ \mathit{merge}_{\mathit{field}}\ \mathit{merge}_{\mathit{row}}\\
&\mathit{switch}_p &&= \mathit{combine}\ \mathit{switch}_{\mathit{field}}\ \mathit{distribute}_{\mathit{row}}\\
&\mathit{select}_p &&= \mathit{combine}\ \mathit{select}_{\mathit{field}}\ \mathit{distribute}_{\mathit{row}}\\
&\mathit{overlay}_p &&= \mathit{combine}\ \mathit{overlay}_{\mathit{field}}\ \mathit{overlay}_{\mathit{row}}
\end{aligned}
$$

The resulting substitution is the combination of the substitutions $U_4 \circ U_3 \circ U_2 \circ U_1$. This combination is always safe since the sets of variables instantiated by the substitutions are disjoint. This can be proved by considering the structure of the substitutions returned by the operation *extend-record*, and by the corresponding field and row operations.

Note that in *apply-fields*, the resulting substitution U from each use of the field operation is applied to the remainder of the fields before any further pairs are combined. Finally, *combine* combines the row variables using the row operation op_r. Note that, since the processing of the fields cannot instantiate a row variable, the substitution U_3 does not need to be applied to the row variables row_1 and row_2 before applying op_r.

The algorithms for the five field operations are given in Figure 6.4. In the $\mathit{distribute}_{\mathit{field}}$ operation the most important case is the combining of two sort schemes. This relies on the $\mathit{distribute}_{\mathit{sort}}$ operation given in Figure 6.5, which returns the meet of the two sort schemes. Since a sort in canonical form does not contain any meets, the $\mathit{distribute}_{\mathit{sort}}$ operation tries to simplify the result sort scheme to eliminate the meet, if possible. The $\mathit{distribute}_{\mathit{sort}}$ operation is based on the first-order unification algorithm, and it may instantiate (free) sort variables. Since sorts contain meet and join, we could use the associative-commutative (AC) unification algorithm to calculate a

Figure 6.3: Algorithm for *combine*

```
combine    :: (field → field → substitution × field) →
              (row → row → substitution × row) →
              record-scheme → record-scheme →
              substitution × record-scheme

combine $op_f$ $op_r$ $r_1$ $r_2$ =
        let
            $U_1, r_1'$ = extend-record (fields-in $r_2$ − fields-in $r_1$) $r_1$
            $U_2, r_2'$ = extend-record (fields-in $r_1$ − fields-in $r_2$) $r_2$
            $\{fields_1\}row_1 = r_1'$
            $\{fields_2\}row_2 = r_2'$
            $U_3, fields'$ = apply-fields $op_f$ $fields_1$ $fields_2$
            $U_4, row' = op_r\ row_1\ row_2$
        in
            $(U_4 \circ U_3 \circ U_2 \circ U_1, \{fields'\}row')$

where
    fields-in $\{i{:}\ \phi_i\}_{i \in I}$ $[\rho|\gamma] = I$

    extend-record $J$ $\{i{:}\ \phi_i\}_{i \in I}$ = $([\ ], \{i{:}\ \phi_i\}_{i \in I}$ @ $\{j{:}\ \mathbf{absent}\}_{j \in J})$

    extend-record $J$ $\{i{:}\ \phi_i\}_{i \in I}\ \rho$ =
        let
            $r' = \{j{:}\ \Delta_j\}_{j \in J}\ \rho'$
        in
            $([\rho \mapsto r'], \{i{:}\ \phi_i\}_{i \in I}$ @ $r')$
        (where all the $\Delta_j$ and $\rho'$ are fresh)

    extend-record $J$ $\{i{:}\ \phi_i\}_{i \in I}\ \gamma$ =
        let
            $r' = \{j{:}\ \Delta_j\}_{j \in J}\ \gamma'$
        in
            $([\gamma \mapsto r'], \{i{:}\ \phi_i\}_{i \in I}$ @ $r')$
        (where all the $\Delta_j$ and $\gamma'$ are fresh)

    apply-fields $op_{field}$ $fields_1$ $fields_2$ =
        if $fields_1 = fields_2 = \{\ \}$ then $([\ ], \{\ \})$
        else   let
                   $\{i{:}\ \phi_1\}$ @ $fields_1' = fields_1$
                   $\{i{:}\ \phi_2\}$ @ $fields_2' = fields_2$
                   $U, \phi' = op_{field}\ \phi_1\ \phi_2$
                   $U', fields'$ = apply-fields $op_{field}$ $U(fields_1')$ $U(fields_2')$
               in
                   $(U' \circ U, \{\ i{:}\ \phi'\}$ @ $fields')$
```

Figure 6.4: Algorithms for the field operations

distribute$_{field}$	::	field $\rightarrow$ field $\rightarrow$ substitution $\times$ field
merge$_{field}$	::	field $\rightarrow$ field $\rightarrow$ substitution $\times$ field
switch$_{field}$	::	field $\rightarrow$ field $\rightarrow$ substitution $\times$ field
select$_{field}$	::	field $\rightarrow$ field $\rightarrow$ substitution $\times$ field
overlay$_{field}$	::	field $\rightarrow$ field $\rightarrow$ substitution $\times$ field

distribute$_{field}$	σ_i	σ_j	=	*distribute*$_{sort}$ σ_i σ_j
distribute$_{field}$	**absent**	**absent**	=	([ ], **absent**)
distribute$_{field}$	Δ	σ	=	($[\Delta \mapsto \sigma], \sigma$)
distribute$_{field}$	Δ	**absent**	=	([$\Delta \mapsto$ **absent**], **absent**)
distribute$_{field}$	Δ_i	Δ_j	=	if $\Delta_i = \Delta_j$ then ([ ], Δ_i)
				else ($[\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta], \Delta$)
				(where Δ is fresh)

distribute$_{field}$ is commutative and all other cases are *failure.*

merge$_{field}$	ϕ	**absent**	=	([ ], ϕ)
merge$_{field}$	Δ	σ	=	([$\Delta \mapsto$ **absent**], σ)
merge$_{field}$	Δ_i	Δ_j	=	if $\Delta_i = \Delta_j$ then ([$\Delta_i \mapsto$ **absent**], **absent**)
				else *failure*

merge$_{field}$ is commutative and all other cases are *failure.*

switch$_{field}$	σ_i	σ_j	=	([ ], $\sigma_i \mid \sigma_j$)
switch$_{field}$	**absent**	**absent**	=	([ ], **absent**)
switch$_{field}$	Δ	σ	=	($[\Delta \mapsto \sigma], \sigma$)
switch$_{field}$	Δ	**absent**	=	([$\Delta \mapsto$ **absent**], **absent**)
switch$_{field}$	Δ_i	Δ_j	=	if $\Delta_i = \Delta_j$ then ([ ], Δ_i)
				else ($[\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta], \Delta$)
				(where Δ is fresh)

switch$_{field}$ *is* commutative and all other cases are *failure.*

select$_{field}$	σ_i	σ_j	=	([ ], $\sigma_i \mid \sigma_j$)
select$_{field}$	**absent**	**absent**	=	([ ], **absent**)
select$_{field}$	Δ	σ	=	($[\Delta \mapsto \sigma], \sigma$)
select$_{field}$	Δ_i	Δ_j	=	if $\Delta_i = \Delta_j$ then ([ ], Δ_i)
				else ($[\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta], \Delta$)
				(where Δ is fresh)

select$_{field}$ is commutative and all other cases are *failure.*

overlay$_{field}$	**absent**	ϕ	=	([ ], ϕ)
overlay$_{field}$	σ	ϕ	=	([ ], σ)
overlay$_{field}$	Δ_i	Δ_j	=	if $\Delta_i = \Delta_j$ then ([ ], Δ_i) else *failure*

overlay$_{field}$ is *failure* in all other cases.

most-general-unifier (mgu) for sort schemes. In the absence of a principal sort, however, we do not require an mgu, and so the complexity of AC unification can be avoided.

The $\mathit{switch}_{\mathit{field}}$ algorithm is identical to the $\mathit{distribute}_{\mathit{field}}$ algorithm, except in the case where it combines two sort schemes. Here, it simply takes the join of the two sort schemes without trying to simplify. The sort inference algorithm is not affected by the sort not being in canonical form, and in the implementation, the simplification of the sort scheme is performed at a later stage.

The operations $\mathit{merge}_{\mathit{field}}$ and $\mathit{overlay}_{\mathit{field}}$ never combine their two field arguments. These operations just select between one of the input fields. In $\mathit{merge}_{\mathit{field}}$, only one of the fields is allowed to be present, and in $\mathit{overlay}_{\mathit{field}}$ the first argument is given priority over the second. Notice that $\mathit{overlay}_{\mathit{field}}$ cannot instantiate any variables, and always

Figure 6.5: Algorithm for the sort operation

```
distribute_sort  ::  sort-scheme → sort-scheme → substitution × sort-scheme

distribute_sort  $\theta_i$        $\theta_j$        =  if $\theta_i = \theta_j$ then ([ ], $\theta_i$)
                                                 else ([$\theta_i \mapsto \theta$, $\theta_j \mapsto \theta$], $\theta$)
                                                 (where $\theta$ is fresh)
distribute_sort  $\theta$          $\sigma$          =  ([$\theta \mapsto \sigma$], $\sigma$)          (where $\sigma \neq \theta_i$)
distribute_sort  $C[\sigma_i]$     $C[\sigma_j]$     =  let
                                                     $U, \sigma' = \mathit{distribute}_{sort}\ \sigma_i\ \sigma_j$
                                                 in
                                                     $(U, C[\sigma'])$
distribute_sort  $I$          $B$          =  if $I \in B$ then ([ ], $I$) else failure
distribute_sort  $I_i$        $I_j$        =  if $I_i = I_j$ then ([ ], $I_i$) else failure
distribute_sort  $B_i$        $B_j$        =  if $B_i = B_j$ then ([ ], $B_i$) else failure
distribute_sort  nothing      $\sigma$          =  ([ ], nothing)
distribute_sort  datum        $\sigma$          =  ([ ], $\sigma$)
distribute_sort  $(\sigma_i \mid \sigma_j)$   $\sigma$          =  ([ ], $(\sigma_i \mid \sigma_j)$ & $\sigma$)
distribute_sort  $(\sigma_i$ & $\sigma_j)$   $\sigma$          =  let
                                                     $U_i, \sigma_i' = \mathit{distribute}_{sort}\ \sigma_i\ \sigma$
                                                     $U_j, \sigma_j' = \mathit{distribute}_{sort}\ U_i(\sigma_j)\ U_i(\sigma)$
                                                 in
                                                     $(U_j \circ U_i,\ U_j(\sigma_i')$ & $\sigma_j')$
distribute_sort is commutative and all other cases are failure.
```

Figure 6.6: Algorithms for the row operations

$distribute_{row}$	::	row → row → substitution × row		
$merge_{row}$	::	row → row → substitution × row		
$overlay_{row}$	::	row → row → substitution × row		
$distribute_{row}$	**exactly**	**exactly**	=	([ ], **exactly**)
$distribute_{row}$	**exactly**	ρ	=	([$\rho \mapsto$ { }], **exactly**)
$distribute_{row}$	**exactly**	γ	=	([$\gamma \mapsto$ { }], **exactly**)
$distribute_{row}$	ρ_i	ρ_j	=	if $\rho_i = \rho_j$ then ([ ], ρ_i) else ([$\rho_i \mapsto$ { }ρ, $\rho_j \mapsto$ { }ρ], ρ) (where ρ is fresh)
$distribute_{row}$	γ_i	γ_j	=	if $\gamma_i = \gamma_j$ then ([ ], γ_i) else ([$\gamma_i \mapsto$ { }γ, $\gamma_j \mapsto$ { }γ], γ) (where γ is fresh)
$distribute_{row}$	ρ	γ	=	([$\gamma \mapsto$ { }ρ], ρ)

distribute$_{row}$ is commutative.

$merge_{row}$	**exactly**	**exactly**	=	([ ], **exactly**)
$merge_{row}$	**exactly**	ρ	=	([ ], ρ)
$merge_{row}$	ρ_i	ρ_j	=	if $\rho_i = \rho_j$ then ([$\rho_i \mapsto$ { }], **exactly**) else *failure*

merge$_{row}$ is commutative and all other cases are *failure*.

$overlay_{row}$	**exactly**	**exactly**	=	([ ], **exactly**)
$overlay_{row}$	**exactly**	ρ	=	([ ], ρ)
$merge_{row}$	ρ_i	ρ_j	=	if $\rho_i = \rho_j$ then ([ ], ρ_i) else *failure*

overlay$_{row}$ is *failure* in all other cases.

returns an empty substitution. In this case, the substitution parameter could be eliminated from the operation, but this would destroy the uniformity of the field operations, and complicate the overall algorithm for the auxiliary operations.

The algorithms for the three row operations are given in Figure 6.6. Here we use the notation "**exactly**" to denote the absence of a row variable, since the record sort scheme contains *exactly* these fields, and no others.

The *distribute*$_{row}$ operation is the only row operation that has to allow for γ-variables, since it is the only one that is used on input record sort schemes. The operations *merge*$_{row}$ and *overlay*$_{row}$ are only used on output record sort schemes.

Consider again the following example of the *distribute* operation from Section 6.2.1:

- *distribute* $\{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ \sigma_2\}\rho_1\ \{\mathrm{x}{:}\ \sigma_3, \mathrm{y}{:}\ \sigma_4\}\rho_2$

Using the algorithm of Figures 6.2 to 6.6, the calculation would proceed as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{distribute}_p\ \{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ \sigma_2\}\rho_1\ \{\mathrm{x}{:}\ \sigma_3, \mathrm{y}{:}\ \sigma_4\}\rho_2 =} \\
 & \mathit{combine}\ \mathit{distribute}_{\mathit{field}}\ \mathit{distribute}_{\mathit{row}}\ \{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ \sigma_2\}\rho_1\ \{\mathrm{x}{:}\ \sigma_3, \mathrm{y}{:}\ \sigma_4\}\rho_2 \\
r_1 & = \{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ \sigma_2\}\rho_1 \\
r_2 & = \{\mathrm{x}{:}\ \sigma_3, \mathrm{y}{:}\ \sigma_4\}\rho_2 \\
U_1, r_1' & = \mathit{extend\text{-}record}\ r_1\ \{\mathrm{y}\} \\
 & = [\rho_1 \mapsto \{\mathrm{y}{:}\ \Delta_1\}\rho_3], \{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ \sigma_2, \mathrm{y}{:}\ \Delta_1\}\rho_3 \\
U_2, r_2' & = \mathit{extend\text{-}record}\ r_2\ \{\mathrm{w}\} \\
 & = [\rho_2 \mapsto \{\mathrm{w}{:}\ \Delta_2\}\rho_4], \{\mathrm{w}{:}\ \Delta_2, \mathrm{x}{:}\ \sigma_3, \mathrm{y}{:}\ \sigma_4\}\rho_4 \\
U_3, \mathit{fields}' & = \mathit{apply\text{-}fields}\ \mathit{distribute}_{\mathit{field}}\ \{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ \sigma_2, \mathrm{y}{:}\ \Delta_1\}\ \{\mathrm{w}{:}\ \Delta_2, \mathrm{x}{:}\ \sigma_3, \mathrm{y}{:}\ \sigma_4\} \\
 & = [\Delta_1 \mapsto \sigma_4, \Delta_2 \mapsto \sigma_1], \{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ (\sigma_2\ \&\ \sigma_3), \mathrm{y}{:}\ \sigma_4\} \\
U_4, \mathit{row}' & = \mathit{distribute}_{\mathit{row}}\ \rho_3\ \rho_4 \\
 & = [\rho_3 \mapsto \{\ \}\rho_5, \rho_4 \mapsto \{\ \}\rho_5], \rho_5
\end{array}$$

So the resulting record sort scheme is $\{\mathrm{w}{:}\ \sigma_1, \mathrm{x}{:}\ (\sigma_2\ \&\ \sigma_3), \mathrm{y}{:}\ \sigma_4\}\rho_5$ and the final substitution is $[\rho_3 \mapsto \{\ \}\rho_5,\ \rho_4 \mapsto \{\ \}\rho_5,\ \Delta_1 \mapsto \sigma_4,\ \Delta_2 \mapsto \sigma_1,\ \rho_2 \mapsto \{\mathrm{w}{:}\ \Delta_2\}\rho_4,$ $\rho_1 \mapsto \{\mathrm{y}{:}\ \Delta_1\}\rho_3]$. This is consistent with our previous example, although the algorithm introduces more variables.

6.2.6 Algebraic Properties of the Auxiliary Operations

In this section, we consider the algebraic properties of the auxiliary operations. These properties are important for reasoning about the sort inference rules—both for proving the correctness of individual rules, and for proving the soundness of the sort inference algorithm. The algebraic properties we are interested in are commutativity, associativity, idempotency, simplification, and ordering. Initially, we will consider the first three of these.

It is straightforward to show that the operations *combine*, *extend-record* and *apply-fields* do not affect the algebraic properties of the auxiliary operations. The algebraic properties of the auxiliary operations come directly from the corresponding properties of the field and row operations they use. Also, since the auxiliary operations model the behaviour of the action combinators, we would expect the operations to possess the same algebraic properties as the action combinators that use them.

The only problem we have is the allocation of fresh variables that occurs in the algorithms. Since different applications of an operation may result in different variables being used, we will consider two record sort schemes to be equal if there is a simple renaming of variables that transforms one into the other. The algebraic properties of the operations are summarised in Table 6.2.

Table 6.2: Algebraic properties of the auxiliary operations

	distribute	*merge*	*switch*	*select*	*overlay*
commutative	✔	✔	✔	✔	✘
associative	✔	✔	✔	✔	✔
idempotent†	✔	✘	✔	✔	✔

†$f\,x\,x = x$ for all x.

Intuitively, *overlay* cannot be commutative, since overlaying is not a commutative operation. Also, *merge* cannot be idempotent, since merging a record sort scheme with itself must fail due to the overlapping domains of the records. Commutativity follows directly from the definitions of the field operations. Associativity and idempotency can be proved by considering all of the possible combinations of arguments to the corresponding field operations.

Next, let us consider the simplification of the auxiliary operations. Often one of the arguments to an auxiliary operation is "empty", i.e. either $\{\ \}\gamma$ for an input record sort scheme, or $\{\ \}$ for an output record sort scheme. The result of an auxiliary operation applied to such an "empty" argument can always be simplified according to the following laws (assuming the operation does not fail):

- *distribute* $\{\ \}\gamma\ \beta = \beta$, since $\{\ \}\gamma$ will add no additional fields to the result.
- *switch* $\{\ \}\gamma\ \beta = \beta$, since $\{\ \}\gamma$ will add no additional fields to the result.
- *merge* $\{\ \}\ \beta = \beta$, since $\{\ \}$ will add no additional fields to the result, and cannot lead to overlapping fields.
- *overlay* $\{\ \}\ \beta = \beta$, since $\{\ \}$ will add no additional fields to the result.
- *select* $\{\ \}\ \beta = \{\ \}$, since the result must have the same domain as its arguments.

Finally, let us consider the ordering that exists between the input record sort schemes and the output record sort scheme for the auxiliary operations. We define the ordering of record sort schemes in the standard way, namely $\beta_i \sqsubseteq \beta_j$ if and only if β_i contains at least the fields of β_j, and each field in β_i is a subsort of the corresponding field in β_j (similarly for $\tau_i \sqsubseteq \tau_j$). Also, since a record sort scheme may contain uninstantiated variables, then the ordering must hold for *all* possible instantiations of those variables.

We formalise the ordering relationship for the auxiliary operations in Chapter 7, as part of the proof of soundness. For now however, we intuitively expect the following relationships to hold. Although the properties are stated using binding record sort schemes, they also hold for transient record sort schemes.

- *distribute* $\beta_1\ \beta_2 \sqsubseteq \beta_1$, since *distribute* $\beta_1\ \beta_2$ contains at least as many fields as β_1 and we take the meet of overlapping fields. This relies on the fact that, for all sorts S_1 and S_2, $S_1\ \&\ S_2 \leq S_1$.

- *merge* $\beta_1\ \beta_2 \sqsubseteq \beta_1$, since *merge* $\beta_1\ \beta_2$ contains at least as many fields as β_1 and corresponding fields are identical.

- $\beta_1 \sqsubseteq$ *select* $\beta_1\ \beta_2$, since *select* $\beta_1\ \beta_2$ contains the same fields as β_1 and we take the join of overlapping fields. This relies on the fact that, for all sorts S_1 and S_2, $S_1 \leq S_1 \mid S_2$.

Here, we are assuming that a particular auxiliary operation does not fail, and that the resulting substitution is applied to the input record sort schemes, β_1 and β_2, as well as the output record sort scheme. Also, since all of these operations are commutative, the ordering holds for the second argument as well as the first.

6.3 Sort Inference Algorithm

Our sort inference algorithm is based on the Even–Schmidt algorithm [ES90], but is improved in several important respects. Our algorithm achieves a greater measure of internal uniformity, by using record schemes for both transients and bindings. It infers exactly which transients and bindings an action uses, using γ-variables to represent transients and bindings passed to the action but not used. It infers action sorts more precisely, by using a more refined sort hierarchy. Not least, it handles a much larger and more representative subset of action notation, including choice, iteration, and abstractions, all of which are essential for writing useful action-semantic descriptions.

6.3.1 Sort Inference Rules

Using the action sort notation introduced in Chapter 5, we begin by specifying the structure of the sort inference rules. We use the following judgements for assigning a sort to an action A and a yielder Y, respectively:

$$\varepsilon \vdash A : (\tau, \beta) \hookrightarrow (\tau', \beta')$$

$$\varepsilon \vdash Y : (\tau, \beta) \rightsquigarrow \sigma$$

where $\varepsilon ::= \{i : (\sigma_1,\ldots,\sigma_n) \to \sigma_i\}_{i \in I}$ with $n \geq 0$ and I a finite subset of Symbol.

Here ε is the *sort environment*, containing (among other things), sort information about constants such as false, true, truth-value, and integer, and about operations such as sum(_,_). It is used to determine the sort of a symbol (a constant or operation name) occurring in an action. The sort environment is essentially fixed, as action notation does not allow new symbols to be introduced[1]. The only modification of the sort environment is in the sort inference rule for "unfolding", where the sort environment is used to propagate the corresponding action sort to the enclosed "unfold" actions.

A sort inference rule consists of a (possibly empty) set of antecedents separated by semicolons, and a single conclusion. An antecedent may be a sort judgement or a *constraint*. A constraint is a side-condition that must hold for the rule to be valid. All of our constraints are of the form "σ_i & $\sigma_j \neq$ nothing" for sort schemes σ_i and σ_j. The conclusion is always a sort judgement and is separated from the antecedents by a horizontal rule. The general form is:

(RULE NAME)
$$\frac{\mathit{Antecedent}_1 \ ; \ \mathit{Antecedent}_2 \ ; \ \ldots \ ; \ \mathit{Antecedent}_n}{\mathit{Conclusion}}$$

Any free variables (i.e. θ_i, Δ_i, ρ_i, and γ_i) occurring in a sort inference rule are assumed to be freshly allocated. In effect, the sort assigned to an action could be universally quantified over these free variables. For example, the sort assigned to "complete" could be "$\forall\gamma_i,\gamma_j$ s.t. $\gamma_i \neq \gamma_j$, $(\{\ \}\gamma_i,\ \{\ \}\gamma_j) \hookrightarrow (\{\ \},\ \{\ \})$". However such quantified schemes are not first class objects in our system, and, in practice, any such sort scheme is immediately instantiated. For this reason, our sort inference algorithm does not contain rules for the introduction and elimination of universal quantification. However, action sorts are still polymorphic in the traditional sense.

1. New symbols can be introduced only using the meta-notation.

If a sort inference rule contains an application of an auxiliary operation, and that operation fails, then rule is invalid and the action (or yielder) is ill-sorted. An ill-sorted action (or yielder) is assigned the sort "nothing".

In the following sections, we consider the sort inference rules for each facet in turn. We have omitted the (trivial) rules for mapping a (syntactic) sort S into its corresponding sort scheme σ. The complete set of inference rules is given in Appendix B.

6.3.2 Basic Action Notation

The following are some of the rules for basic actions and combinators:

(COMPLETE-I)

$$\frac{}{\varepsilon \vdash \text{complete} : (\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{\ \}, \{\ \})}$$

(AND-I)

$$\frac{\varepsilon \vdash A_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1') \ ; \ \varepsilon \vdash A_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{c}\varepsilon \vdash A_1 \text{ and } A_2 : (\textit{distribute}\ \tau_1\ \tau_2, \textit{distribute}\ \beta_1\ \beta_2) \hookrightarrow \\ (\textit{merge}\ \tau_1'\ \tau_2', \textit{merge}\ \beta_1'\ \beta_2')\end{array}}$$

(OR-I)

$$\frac{\varepsilon \vdash A_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1') \ ; \ \varepsilon \vdash A_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{c}\varepsilon \vdash A_1 \text{ or } A_2 : (\textit{switch}\ \tau_1\ \tau_2, \textit{switch}\ \beta_1\ \beta_2) \hookrightarrow \\ (\textit{select}\ \tau_1'\ \tau_2', \textit{select}\ \beta_1'\ \beta_2')\end{array}}$$

Rule (COMPLETE-I) is trivial. The primitive action "complete" accepts arbitrary transients and bindings and produces empty transients and bindings.

Rule (AND-I) illustrates the typical structure of a sort inference rule for an action combinator. The sorts of the sub-actions A_1 and A_2 are combined to produce the sort of the whole action. Since "and" distributes its received information and merges its produced information, we use the *distribute* operation on the input transients and bindings, and the *merge* operation on the output transients and bindings.

Rule (OR-I) is structurally similar to rule (AND-I). The only difference is that *distribute* is replaced by *switch*, and *merge* is replaced by *select* reflecting the different data flows used by the "or" action.

The following rules show how we infer the sorts of "unfolding" actions:

(UNFOLDING-I)

$$\frac{[\text{unfold} : (\tau, \beta) \hookrightarrow (\tau', \beta')]\ \varepsilon \vdash A : (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \text{unfolding}\ A : (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

(UNFOLD-I)

$$\frac{}{[\text{unfold} : (\tau, \beta) \hookrightarrow (\tau', \beta')]\ \varepsilon \vdash \text{unfold} : (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

We insist that, inside "unfolding *A*", every occurrence of "unfold" has the same sort $(\tau, \beta) \hookrightarrow (\tau', \beta')$, which is the sort of *A* itself. This restriction excludes polymorphic "unfolding" actions[2]. However, it does not exclude the "unfolding" actions that occur in practical situations, such as specification of the semantics of loops in programming languages.

The simplicity of the sort inference rules (UNFOLDING) and (UNFOLD) belies their true power. For example, unlike Palsberg's subset of action notation[Pal92b], we do not restrict "unfolding" actions to be tail-recursive. This decision (and the use of abstractions) is the main complicating factor in our sort inference algorithm. It prevents us from using either a simple bottom-up or top-down analysis to infer the sort of an action, since we cannot determine the sort of an enclosed "unfold" action without knowing the sort of the entire action.

2. For which sort inference is undecidable [Sch91].

6.3.3 Functional Action Notation

The following are the most important rules that deal with transients:

(GIVE-I)

$$\frac{\varepsilon \vdash Y : (\tau, \beta) \rightsquigarrow \sigma}{\varepsilon \vdash \text{give } Y \text{ label \# } n : (\tau, \beta) \hookrightarrow (\{n : \sigma\}, \{\ \})}$$

(THE-I)

$$\frac{\varepsilon \vdash S : \sigma\ ;\ \ \theta\ \&\ \sigma \neq \text{nothing}}{\varepsilon \vdash \text{the } S \text{ \# } n : (\{n : \theta\}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow (\theta\ \&\ \sigma)}$$

(IT-I)

$$\frac{}{\varepsilon \vdash \text{it} : (\{0 : \theta\}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \theta}$$

(THEN-I)

$$\frac{\varepsilon \vdash A_1 : (\tau_1, \beta_1) \hookrightarrow (\tau, \beta_1')\ ;\ \ \varepsilon \vdash A_2 : (\tau, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash A_1 \text{ then } A_2 : (\tau_1, \textit{distribute}\ \beta_1\ \beta_2) \hookrightarrow (\tau_2', \textit{merge}\ \beta_1'\ \beta_2')}$$

Rule (GIVE-I) illustrates a primitive action that contains a yielder. Having inferred the sort of the yielder *Y*, it is straightforward to construct the sort of "give *Y* label # *n*". Since "give" does not need any more information, the input transient and binding sorts are the same as those of the yielder *Y*, and the sort of the output transient is just the sort σ returned by the yielder.

The action "give *Y*" is an abbreviation for the action "give *Y* label # 0" and so does not require its own rule.

Rule (THE-I) infers that the yielder "the *S* # *n*" expects to receive a transient labelled *n* of sort θ. Here θ is a *sort variable*, which is to be instantiated to some actual sort that satisfies the stated constraint that "θ & σ ≠ nothing". The sort scheme σ is the translation of the syntactic sort *S*. The sort variable θ will be instantiated to a particular sort, depending on the received transients. The instantiation of θ can greatly affect the output sort of the yielder. If θ is instantiated to a subsort of σ, then the output sort θ & σ will be more precise than σ. The output sort may even be an individual sort if θ is instantiated to an individual. If, however, θ is instantiated to a supersort of σ, then the inference rule indicates a place where a run-time sort check is required, since the transient labelled n might turn out at run-time not to be of sort σ. The following examples illustrate these different possibilities:

- If σ is (integer | truth-value) and θ is instantiated to integer, then the output sort becomes (integer | truth-value) & integer = integer.
- If σ is (integer | truth-value) and θ is instantiated to 7, then the output sort becomes (integer | truth-value) & 7 = 7.
- If σ is integer and θ is instantiated to (integer | truth-value), then the output sort becomes integer & (integer | truth-value) = integer. Here the received datum must be checked to make sure it is not actually a truth-value.

The yielder "the *S*" is an abbreviation for the yielder "the *S* # 0", and so also does not require its own rule. Although "it" is equivalent to "the datum # 0", its rule is useful, since it eliminates the redundant constraint "θ & datum $\neq$ nothing".

Note that in both the (GIVE-I) and (THE-I) rules, the label n can only be a natural number and not, for example, a yielder of a natural. Therefore, the set of labels appearing in an action can statically determined. The standard action notation does permit the label n to be the result of evaluating a yielder, but this was not the case with the earlier version of action notation used as the basis of ACTRESS action notation.

In rule (THEN-I), the record sort scheme τ is used for both the output transients from A_1 and the input transients to A_2. In practice, this insists that the sort of transients produced by A_1 be unified with the sort of transients received by A_2. If τ_1' is the transient sort scheme produced by A_1, and τ_2 is the transient sort scheme required by A_2 then some examples of the unification of two record sort schemes are:

- $\tau_1' = \{1\colon \sigma_1, 2\colon \sigma_2\}$ and $\tau_2 = \{1\colon \sigma_1, 2\colon \sigma_3\}$. These can be made equal by replacing the sorts of the overlapping fields by their meet in both τ_1' and τ_2, $\{1{:}\ \sigma_1, 2{:}\ (\sigma_2 \& \sigma_3)\}$. If $\sigma_2 \& \sigma_3 =$ nothing, τ_1' and τ_2 cannot be made equal, therefore we have inferred that "A_1 then A_2" is ill-sorted. To be concrete, if $\sigma_2 = 7$ and $\sigma_3 =$ integer, then $\sigma_2 \& \sigma_3 = 7$; in other words, having already inferred that A_2 expects an unknown transient of sort integer, we have now inferred from the context of A_2 that the integer is, in fact, 7. Or if $\sigma_2 =$ truth-value and $\sigma_3 =$ integer, then $\sigma_2 \& \sigma_3 =$ nothing; in other words, A_1 gives a truth value, but A_2 expects a transient of sort integer; clearly "A_1 then A_2" is ill-sorted.

- $\tau_1' = \{1{:}\ \sigma_1, 2{:}\ \sigma_2\}$ and $\tau_2 = \{1{:}\ \sigma_1\}\gamma_1$. These can be made equal by instantiating γ_1 to $\{2{:}\ \sigma_2\}$. In other words, we have inferred that A_2 receives a transient, labelled 2, of sort σ_2 (which it ignores) as well as a transient, labelled 1, of sort σ_1.

- $\tau_1' = \{1{:}\ \sigma_1, 2{:}\ \sigma_2\}$ and $\tau_2 = \{1{:}\ \sigma_1, 3{:}\ \sigma_3\}\gamma_1$. These cannot be made equal, however we instantiate γ_1. Action A_2 requires a transient, labelled 3, of sort σ_3, which is not given by action A_1. Therefore we have inferred that "A_1 then A_2" is ill-sorted.

6.3.4 Declarative Action Notation

The following are the most important rules that deal with bindings:

(BIND-I)

$$\frac{\varepsilon \vdash Y : (\tau, \beta) \rightsquigarrow \sigma\ ;\ \text{bindable} \ \&\ \sigma \neq \text{nothing}}{\varepsilon \vdash \mathsf{bind}\ k\ \mathsf{to}\ Y : (\tau, \beta) \hookrightarrow (\{\ \}, \{k : \sigma\})}$$

(BOUND-I)

$$\frac{\varepsilon \vdash S : \sigma\ ;\ \theta \ \&\ \sigma \neq \text{nothing}}{\varepsilon \vdash \mathsf{the}\ S\ \mathsf{bound\ to}\ k : (\{\ \}\gamma_1, \{k : \theta\}\gamma_2,) \rightsquigarrow \theta \ \&\ \sigma}$$

(HENCE-I)

$$\frac{\varepsilon \vdash A_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta)\ ;\ \varepsilon \vdash A_2 : (\tau_2, \beta) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash A_1 \ \textsf{hence}\ A_2 : (\textit{distribute}\ \tau_1\ \tau_2, \beta_1) \hookrightarrow (\textit{merge}\ \tau_1'\ \tau_2', \beta_2')}$$

(MOREOVER-I)

$$\frac{\varepsilon \vdash A_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1')\ ;\ \varepsilon \vdash A_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{l}\varepsilon \vdash A_1 \ \textsf{moreover}\ A_2 : (\textit{distribute}\ \tau_1\ \tau_2, \textit{distribute}\ \beta_1\ \beta_2) \hookrightarrow \\ \qquad\qquad (\textit{merge}\ \tau_1'\ \tau_2', \textit{overlay}\ \beta_2'\ \beta_1')\end{array}}$$

Rule (BIND-I) is straightforward. Rule (BOUND-I) is analogous to rule (THE-I) for transients. Note that, in both of these rules, the binding is always to a known token *k*. This differs from the standard action notation, where the token may be produced by a yielder, and hence may only be discovered when the action is performed. It is, however, consistent with the earlier version of action notation that was used as the basis for ACTRESS action notation. This restriction is necessary to allow us to use record sort schemes to model the declarative facet, but this does not restrict the typical action used to denote a program, where the set of identifiers used in the program is static.

Rule (HENCE-I) in the declarative facet is analogous to rule (THEN-I) in the functional facet. Rule (MOREOVER-I) is almost identical to rule (AND-I): the only difference is the use of *overlay* rather than *merge* for the output binding record scheme.

The remaining two declarative action combinators, "furthermore" and "before", represent the most complex data flows found in ACTRESS action notation, and give us the only rules that contain ρ-variables. "furthermore" is an abbreviation, and so its rule can be derived from the rules of its component actions. "before" can be approximated by other notation, and so its rule can be justified in terms of other inference rules. We consider these combinators in the following two sections.

6.3.4.1 Sort inference rule for "furthermore"

The sort inference rule for "furthermore" is:

(FURTHERMORE-I)

$$\frac{\varepsilon \vdash A : (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \textsf{furthermore}\ A : (\tau, \textit{distribute}\ \{\ \}\rho\ \beta) \hookrightarrow (\tau', \textit{overlay}\ \beta'\ \{\ \}\rho)}$$

The action "furthermore *A*" is polymorphic in the sorts of its bindings. Here, the output bindings are the same as those received by the action, except that they are *overlaid* by any bindings produced by *A*. The record sort scheme "{ }ρ" represents the (unknown) bindings propagated by "furthermore".

Since "furthermore *A*" is an abbreviation for "rebind moreover *A*", it is possible to derive its inference rule from the rules for "rebind" and "moreover". Although "rebind" is not part of the ACTRESS subset (see Section 6.2.4 for the reason), its rule would be:

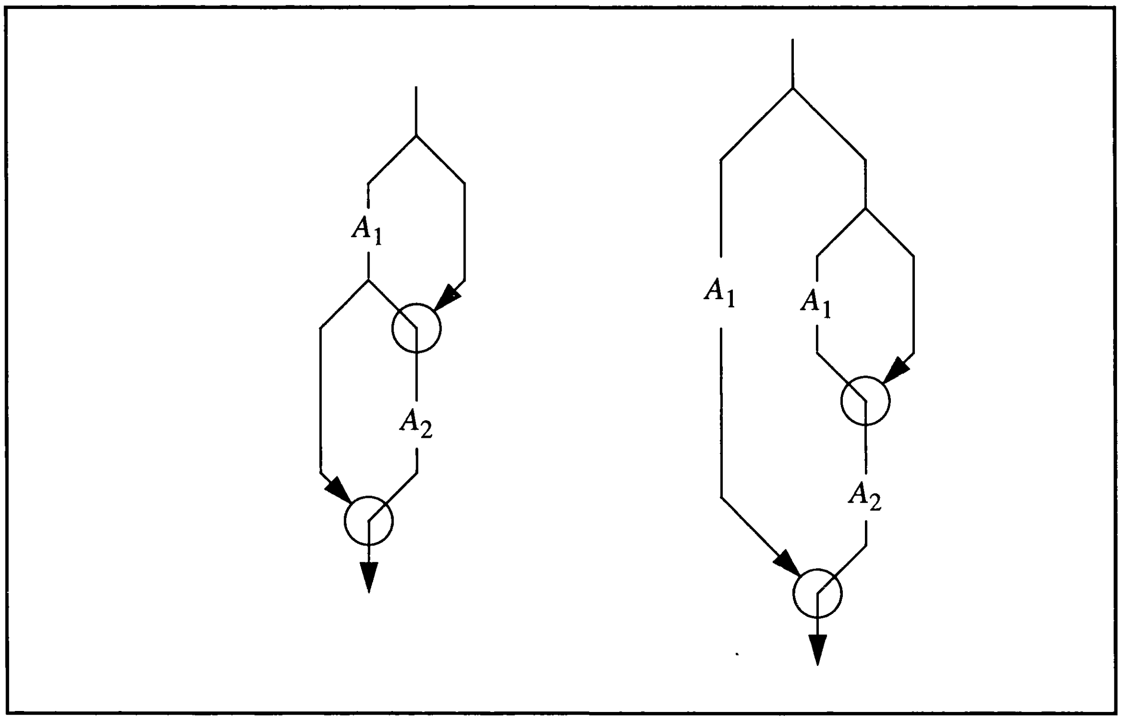
(REBIND-I)

$$\frac{}{\varepsilon \vdash \textsf{rebind} : (\{\ \}\gamma, \{\ \}\rho) \hookrightarrow (\{\ \}, \{\ \}\rho)}$$

So, we have:

$$\cfrac{\cfrac{\cfrac{\varepsilon \vdash \textsf{rebind} : (\{\ \}\gamma, \{\ \}\rho) \hookrightarrow (\{\ \}, \{\ \}\rho)\ ;\ \ \varepsilon \vdash A : (\tau, \beta) \hookrightarrow (\tau', \beta')}{\begin{array}{l}\varepsilon \vdash \textsf{rebind moreover}\ A : (\textit{distribute}\ \{\ \}\gamma\,\tau, \textit{distribute}\ \{\ \}\rho\ \beta) \hookrightarrow \\ \qquad (\textit{merge}\ \{\ \}\ \tau', \textit{overlay}\ \beta'\ \{\ \}\rho)\end{array}}}{\begin{array}{l}\varepsilon \vdash \textsf{furthermore}\ A : (\textit{distribute}\ \{\ \}\gamma\,\tau, \textit{distribute}\ \{\ \}\rho\ \beta) \hookrightarrow \\ \qquad (\textit{merge}\ \{\ \}\ \tau', \textit{overlay}\ \beta'\ \{\ \}\rho)\end{array}}}{\varepsilon \vdash \textsf{furthermore}\ A : (\tau, \textit{distribute}\ \{\ \}\rho\ \beta) \hookrightarrow (\tau', \textit{overlay}\ \beta'\ \{\ \}\rho)}$$

Figure 6.7: Declarative data flows for "A_1 before A_2" and its declarative equivalent



6.3.4.2 Sort inference rule for "before"

The sort inference rule for "before" is:

(BEFORE-I)

$$\frac{\varepsilon \vdash A_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1') \;; \quad \varepsilon \vdash A_2 : (\tau_2, \mathit{overlay}\ \beta_1'\ \{\ \}\rho) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash A_1 \text{ before } A_2 : (\mathit{distribute}\ \tau_1\ \tau_2, \mathit{distribute}\ \{\ \}\rho\ \beta_1) \hookrightarrow (\mathit{merge}\ \tau_1'\ \tau_2', \mathit{overlay}\ \beta_2'\ \beta_1')}$$

The record sort scheme $\{\ \}\rho$ represents the bindings received by the action as a whole which are not overlaid by bindings produced by A_1, and which are required by A_2.

In the declarative facet only, "A_1 before A_2" can be simulated by "A_1 moreover (furthermore A_1 hence A_2)". This provides us with a useful mechanism for verifying the correctness of the inference rule for "before".

The data flows for "A_1 before A_2" and "A_1 moreover (furthermore A_1 hence A_2)" are given in Figure 6.7. The copying of the bindings produced by the action A_1 has been replaced by copying the action A_1 itself. Note that this combination only reflects the flow of bindings between the actions. It is not an equivalence in general, especially as it involves performing A_1 twice.

Since we are verifying the rule for only the declarative behaviour of "before", we will simplify action sorts to the form "$\beta \hookrightarrow \beta'$".

$$\begin{array}{c}
\varepsilon \vdash A_1 : \beta_1 \hookrightarrow \beta'_1 \\
\hline
\varepsilon \vdash \text{furthermore } A_1 : (\mathit{distribute}\ \{\ \}\rho\ \beta_1) \hookrightarrow (\mathit{overlay}\ \beta'_1\ \{\ \}\rho)\ ; \\
\varepsilon \vdash A_2 : \mathit{overlay}\ \beta'_1\ \{\ \}\rho \hookrightarrow \beta'_2 \\
\hline
\varepsilon \vdash \text{furthermore } A_1 \text{ hence } A_2 : (\mathit{distribute}\{\ \}\rho\ \beta_1) \hookrightarrow \beta'_2 \\
\hline
\varepsilon \vdash A_1 \text{ moreover (furthermore } A_1 \text{ hence } A_2) : \\
(\mathit{distribute}\ \beta_1\ (\mathit{distribute}\ \{\ \}\rho\ \beta_1)) \hookrightarrow (\mathit{overlay}\ \beta'_2\ \beta'_1)
\end{array}$$

Since *distribute* is commutative, associative and idempotent, we know that *distribute* β_1 (*distribute* $\{\ \}\rho\ \beta_1) =$ *distribute* $\{\ \}\rho\ \beta_1$, and so the rule becomes:

$$\begin{array}{c}
\varepsilon \vdash A_1 : \beta_1 \hookrightarrow \beta'_1;\ \ \varepsilon \vdash A_2 : \mathit{overlay}\ \beta'_1\ \{\ \}\rho \hookrightarrow \beta'_2 \\
\hline
\varepsilon \vdash \text{furthermore } A_1 \text{ hence } A_2 : (\mathit{distribute}\{\ \}\rho\ \beta_1) \hookrightarrow \beta'_2 \\
\hline
\varepsilon \vdash A_1 \text{ moreover (furthermore } A_1 \text{ hence } A_2) : \\
(\mathit{distribute}\ \{\ \}\rho\ \beta_1) \hookrightarrow (\mathit{overlay}\ \beta'_2\ \beta'_1) \\
\hline
\varepsilon \vdash A_1 \text{ before } A_2 : (\mathit{distribute}\ \{\ \}\rho\ \beta_1) \hookrightarrow (\mathit{overlay}\ \beta'_2\ \beta'_1)
\end{array}$$

6.3.5 Imperative Action Notation

The following are some of the rules that deal with storage:

(STORE-I)

$$\begin{array}{c}
\varepsilon \vdash Y_1 : (\tau_1, \beta_1) \rightsquigarrow \sigma_1\ ; \\
\varepsilon \vdash Y_2 : (\tau_2, \beta_2) \rightsquigarrow \text{cell } [\sigma_2]\ ;\ \sigma_1\ \&\ \sigma_2 \neq \text{nothing} \\
\hline
\varepsilon \vdash \text{store } Y_1 \text{ in } Y_2 : (\mathit{distribute}\ \tau_1\ \tau_2,\ \mathit{distribute}\ \beta_1\ \beta_2) \hookrightarrow (\{\ \},\ \{\ \})
\end{array}$$

(STORED-I)

$$\frac{\varepsilon \vdash S_1 : \sigma_1 \ ; \ \varepsilon \vdash Y_2 : (\tau_2, \beta_2) \rightsquigarrow \text{cell}\ [\sigma_2] \ ; \ \sigma_1 \ \&\ \sigma_2 \neq \text{nothing}}{\varepsilon \vdash \text{the } S_1 \text{ stored in } Y_2 : (\tau_2, \beta_2) \rightsquigarrow \sigma_1 \ \&\ \sigma_2}$$

(ALLOCATE-I)

$$\frac{\varepsilon \vdash S : \text{cell}\ [\sigma]}{\varepsilon \vdash \text{allocate } S : (\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{0 : \text{cell}\ [\sigma]\}, \{\ \})}$$

Since the imperative facet is not directly modelled by our action sorts, the sort inference rules for imperative action notation are imprecise when compared with the rules for functional and declarative action notation. There are, however, a number of constraints that the imperative inference rules can enforce.

Rule (STORE-I) ensures that the sort σ_1 of the datum being stored in a cell is consistent with the sort σ_2 of the datum that the cell can contain. For example, storing an integer in a cell that can contain either an integer or a truth-value will always complete, but storing an integer in a cell that can only contain a truth-value will always fail. Moreover, if the datum could be either an integer or a truth-value, then storing it in a cell that can only contain an integer may or may complete, so we must perform a run-time check to make sure that the datum is an integer.

Rule (STORED-I) is similar to the rules for the other primitive yielders (THE-I) and (BOUND-I). For (STORED-I), however, we cannot represent the sort of datum stored in the cell by a sort variable θ, which is later instantiated to the actual sort by unifying it with the sort of the received storage (as we did with transients and bindings). Here, we can only check that the sort σ_2 of cell yielded by Y_2 is consistent with the sort σ_1.

Rule (ALLOCATE-I) is straightforward.

In the imperative inference rules, the requirement that a yielder yields a datum of sort "cell [σ]" is quite strong. In rules (STORE-I) and (STORED-I), if we only know that the yielder Y_2 produces a datum of sort "integer | cell [integer]", then we will not be able to assign a sort to the action. In practice, this does not appear to be a problem, since cells usually appear as bindings, about which we have precise information. This restriction is somewhat analogous to the one that functional and declarative actions can only use literal labels and tokens. The rules could be written to avoid this restriction, but this would tend to reduce the quality of the sorts inferred for imperative actions. For example, in the yielder "the S_1 stored in Y_2", we would lose information about the sort of cell yielded by Y_2, and this would only allow us to infer that the output was of sort S_1, whereas we currently infer the sort S_1 & S_2, where Y_2 yields a value of sort cell[S_2]. Also, we may have to check at run-time that the datum yielded by Y_2 was actually a cell.

6.3.6 Reflective Action Notation

The following are the rules that deal with abstractions:

(ABSTRACTION-I)

$$\frac{\varepsilon \vdash A : (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \text{abstraction } A : \text{abstraction } (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

(ENACT-I)

$$\frac{\varepsilon \vdash Y : (\tau, \beta) \leadsto (\text{abstraction } (\{\ \}, \{\ \}) \hookrightarrow (\tau'_A, \beta'_A))}{\varepsilon \vdash \text{enact } Y : (\tau, \beta) \hookrightarrow (\tau'_A, \beta'_A)}$$

(CLOSURE-I)

$$\frac{\varepsilon \vdash Y : (\tau, \beta) \leadsto (\text{abstraction } (\tau_A, \beta_A) \hookrightarrow (\tau'_A, \beta'_A))}{\begin{array}{l}\varepsilon \vdash \text{closure } Y : (\tau, \textit{distribute } \beta\ \beta_A) \leadsto \\ \qquad\qquad (\text{abstraction } (\tau_A, \{\ \}) \hookrightarrow (\tau'_A, \beta'_A))\end{array}}$$

(WITH-I)

$$\frac{\begin{array}{c}\varepsilon \vdash Y_1 : (\tau_1, \beta_1) \leadsto (\text{abstraction } (\{0 : \sigma'\}, \beta_A) \hookrightarrow (\tau'_A, \beta'_A))\ ; \\ \varepsilon \vdash Y_2 : (\tau_2, \beta_2) \leadsto \sigma\ ;\ \sigma' \ \&\ \sigma \neq \text{nothing}\end{array}}{\begin{array}{l}\varepsilon \vdash Y_1 \text{ with } Y_2 : (\textit{distribute } \tau_1\ \tau_2, \textit{distribute } \beta_1\ \beta_2) \leadsto \\ \qquad\qquad (\text{abstraction } (\{\ \}, \beta_A) \hookrightarrow (\tau'_A, \beta'_A))\end{array}}$$

Rule (ABSTRACTION-I) shows the isomorphism between the sort of "abstraction A" and the sort of the incorporated action A. The resulting abstraction sort contains the transient scheme τ and binding scheme β that are required by the incorporated action A, and which will be provided by using "with" and "closure" respectively. The transient scheme τ' and binding scheme β' represent the information that will be produced by enacting the incorporated action A, if it completes.

Rule (ENACT-I) insists that the transients and bindings required by the abstraction's incorporated action are empty. Suppose that this is not the case, e.g., that the incorporated action expects to receive non-empty bindings; then the performance of the incorporated action will eventually fail when it tries to use a binding, since "enact A" does not itself supply any bindings to the incorporated action A. (Only the "closure" operation does so.)

Rule (CLOSURE-I) infers the sort of bindings required to form the closure of an abstraction, principally the bindings β_A required by the incorporated action (combined using *distribute* with the bindings β required to evaluate Y). The sort of the resulting abstraction indicates that it requires no bindings (as required for use in an "enact" action).

Rule (WITH-I) is slightly more complicated. Firstly, the incorporated action must expect to receive a single transient datum labelled 0 (the input transient sort scheme must match $\{0 : \sigma'\}$). Secondly, the sort σ' of this transient must be consistent with the sort σ of the datum actually supplied (σ' & $\sigma \neq$ nothing). The sort of the resulting abstraction is made to have empty input transients. This rule is slightly stronger than the standard interpretation, since it makes it impossible to apply "with" to an abstraction twice (on the second application, the abstraction will no longer be expecting the input). In the standard interpretation, this would be a harmless operation.

With these inference rules, abstraction sorts are restricted to being monomorphic, i.e. each application of rules (WITH-I) and (CLOSURE-I), of which there may be several for a particular abstraction, must have the same sort. This is analogous to the problem in the Hindley-Milner type inference algorithm with "λ-bound" versus "let-bound" type schemes. Only let-bound type schemes are universally quantified, and can be instantiated at different places with different types. Since it was previously decided that universally quantified sort schemes were not first-class objects, the elimination of this monomorphic restriction for abstractions would force the introduction of universally-quantified abstraction sort schemes as first-class objects. This would require us to include quantification introduction and elimination rules for abstraction sort schemes (although such rules would still not be required for actions or yielders).

6.4 An Example of Sort Inference

Consider the following little program in a simple imperative language:

```
let  const b ~ true;
     var x: int
in
     while b do
     x := - x
```

This might be mapped to the following program action:

```
furthermore
| | give true then bind "b" to the value
| and then
| | allocate a cell[integer] then bind "x" to the cell
hence
| unfolding
| | | give the value bound to "b" or
| | | give the value stored in the cell bound to "b"
| | then
| | | | | | give the value bound to "x" or
| | | | | | give the value stored in the cell bound to "x"
| | | | | then give negation (the integer)
| | | | | then store the value in the cell bound to "x"
| | | | and then unfold
| | | else complete
```

where the symbols "value" and "cell" represent the sorts "integer | truth-value" and "cel[integer | truth-value]" respectively.

The sort inference algorithm begins with a sort environment ε containing:

> [value: (integer | truth-value), negation(_): integer $\rightarrow$ integer,
>
> cell: cell[integer | truth-value], ...]

and an empty substitution *U*.

First consider the action on line 2. Application of rules (GIVE-I), (BIND-I) and (THE-I) to the sub-actions gives:

$$\dfrac{\dfrac{\varepsilon \vdash \text{true: true}}{\varepsilon \vdash \text{true: } (\{\ \}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \text{true}}}{\varepsilon \vdash \text{give true: } (\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{0\text{: true}\}, \{\ \})}$$

$$\dfrac{\dfrac{\begin{array}{l}\varepsilon \vdash \text{value: integer | truth-value ;}\\ \theta_1 \mathbin{\&} (\text{integer | truth-value}) \neq \text{nothing}\end{array}}{\begin{array}{l}\varepsilon \vdash \text{the value: } (\{0\text{: } \theta_1\}\gamma_3, \{\ \}\gamma_4) \rightsquigarrow \theta_1 \mathbin{\&} (\text{integer | truth-value}) \text{ ;}\\ \text{bindable} \mathbin{\&} (\theta_1 \mathbin{\&} (\text{integer | truth-value})) \neq \text{nothing}\end{array}}}{\varepsilon \vdash \text{bind "b" to the value: } (\{0\text{: } \theta_1\}\gamma_3, \{\ \}\gamma_4) \hookrightarrow (\{\ \}, \{\text{b: } \theta_1 \mathbin{\&} (\text{integer | truth-value})\})}$$

Application of rule (THEN-I) now forces unification of the first sub-action's outgoing transient sort scheme {0: true} with the second sub-action's incoming transient sort scheme $\{0\text{: } \theta_1\}\gamma_3$. Thus the sort variable θ_1 is instantiated to true and γ_3 to { }. The resulting sort assignments are:

$$\dfrac{\begin{array}{l}\varepsilon \vdash \text{give true: } (\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{0\text{: true}\}, \{\ \}) \text{ ;}\\ \varepsilon \vdash \text{bind "b" to the value: } (\{0\text{: true}\}, \{\ \}\gamma_4) \hookrightarrow (\{\ \}, \{\text{b: true}\})\end{array}}{\varepsilon \vdash \text{give true then bind "b" to the value: } (\{\ \}\gamma_1, \{\ \}\gamma_5) \hookrightarrow (\{\ \}, \{\text{b: true}\})}$$

where $U = [\gamma_2 \mapsto \{\ \}\gamma_5, \gamma_4 \mapsto \{\ \}\gamma_5, \theta_1 \mapsto \text{true}, \gamma_3 \mapsto \{\ \}]$

Similarly, the actions on line 4 are assigned the following sorts:

$$\frac{\varepsilon \vdash \text{cell[integer]: cell[integer]}}{\varepsilon \vdash \text{allocate a cell[integer]: } (\{\ \}\gamma_6, \{\ \}\gamma_7) \hookrightarrow (\{0\text{: cell[integer]}\}, \{\ \})}$$

$$\begin{array}{l}
\varepsilon \vdash \text{cell: cell[integer | truth-value] ;} \\
\theta_2\ \&\ (\text{cell[integer | truth-value]}) \neq \text{nothing} \\
\hline
\varepsilon \vdash \text{the cell: } (\{0\text{: } \theta_2\}\gamma_8, \{\ \}\gamma_9) \rightsquigarrow \theta_2\ \&\ (\text{cell[integer | truth-value]}) \text{ ;} \\
\text{bindable}\ \&\ (\theta_2\ \&\ (\text{cell[integer | truth-value]})) \neq \text{nothing} \\
\hline
\varepsilon \vdash \text{bind "x" to the cell: } (\{0\text{: } \theta_2\}\gamma_8, \{\ \}\gamma_9) \hookrightarrow \\
\qquad (\{\ \}, \{\text{x: } \theta_2\ \&\ (\text{cell[integer | truth-value]})\})
\end{array}$$

and the composite action:

$$\begin{array}{l}
\varepsilon \vdash \text{allocate a cell[integer]: } (\{\ \}\gamma_6, \{\ \}\gamma_7) \hookrightarrow (\{0\text{: cell[integer]}\}, \{\ \}) \text{ ;} \\
\varepsilon \vdash \text{bind "x" to the cell: } (\{0\text{: cell[integer]}\}\gamma_8, \{\ \}\gamma_9) \hookrightarrow (\{\ \}, \{0\text{: cell[integer]}\}) \\
\hline
\varepsilon \vdash \text{allocate a cell[integer] then bind "x" to the cell :} \\
\qquad (\{\ \}\gamma_6, \{\ \}\gamma_{10}) \hookrightarrow (\{\ \},\{\text{x: cell[integer]}\})
\end{array}$$

where U is extended with the substitution:

$$[\gamma_7 \mapsto \{\ \}\gamma_{10}, \gamma_9 \mapsto \{\ \}\gamma_{10}, \theta_2 \mapsto \text{cell[integer]}, \gamma_8 \mapsto \{\ \}]$$

The "and then" action on lines 2–4 is assigned the following sort:

$$\begin{array}{l}
\varepsilon \vdash \text{give true then bind "b" to the value: } (\{\ \}\gamma_1, \{\ \}\gamma_5) \hookrightarrow (\{\ \}, \{\text{b: true}\}) \text{ ;} \\
\varepsilon \vdash \text{allocate a cell[integer] then bind "x" to the cell :} \\
\qquad (\{\ \}\gamma_6, \{\ \}\gamma_{10}) \hookrightarrow (\{\ \},\{\text{x: cell[integer]}\}) \\
\hline
\varepsilon \vdash \text{... and then ... : } (\{\}\gamma_{11}, \{\}\gamma_{12}) \hookrightarrow (\{\ \}, \{\text{b: true, x: cell[integer]}\})
\end{array}$$

where U is extended with the substitution:

$$[\gamma_1 \mapsto \{\ \}\gamma_{11}, \gamma_6 \mapsto \{\ \}\gamma_{11}, \gamma_5 \mapsto \{\ \}\gamma_{12}, \gamma_{10} \mapsto \{\ \}\gamma_{12}]$$

We apply rules (BOUND-I), (STORED-I), and (GIVE-I) to the actions on lines 7 and 8:

$\varepsilon \vdash$ value: integer | truth-value

$\varepsilon \vdash$ the value bound to "b": $(\{\}\gamma_{13}, \{\text{b}: \theta_3\}\gamma_{14}) \rightsquigarrow (\theta_3$ & (integer | truth-value)) ;
θ_3 & (integer | truth-value) $\neq$ nothing

$\varepsilon \vdash$ give the value bound to "b":
$(\{\}\gamma_{13}, \{\text{b}: \theta_3\}\gamma_{14}) \hookrightarrow (\{0: (\theta_3$ & (integer | truth-value))$\}, \{\ \})$

$\varepsilon \vdash$ cell: cell[integer | truth-value] ;
θ_4 & cell[integer | truth-value] $\neq$ nothing

$\varepsilon \vdash$ the cell bound to "b": $(\{\ \}\gamma_{15}, \{\text{b}: \theta_4\}\gamma_{16}) \rightsquigarrow (\theta_4$ & cell[integer | truth-value]) ;
$\varepsilon \vdash$ value: integer | truth-value ;
θ_4 & cell[integer | truth-value] = cell[θ_5] ;
θ_5 & (integer | truth-value) $\neq$ nothing

$\varepsilon \vdash$ the value stored in the cell bound to "b":
$(\{\ \}\gamma_{15}, \{\text{b}: \theta_4\}\gamma_{16}) \rightsquigarrow \theta_5$ & (integer | truth-value)

$\varepsilon \vdash$ give the value stored in the cell bound to "b":
$(\{\ \}\gamma_{15}, \{\text{b}: \theta_4\}\gamma_{16}) \hookrightarrow (\{0: (\theta_5$ & (integer | truth-value))$\}, \{\ \})$

subject to the constraints θ_3 & (integer | truth-value) $\neq$ nothing, θ_4 & cell[integer | truth-value] $\neq$ nothing, and θ_5 & (integer | truth-value) $\neq$ nothing. The antecedent "θ_4 & cell[integer | truth-value] = cell[θ_5]" is necessary since the rule (STORED-I) requires the sort of its yielder to be "cell[σ_2]".

Application of rule (OR-I) to the action on lines 7–8 now gives:

$\varepsilon \vdash$ give the value bound to "b":
$(\{\}\gamma_{13}, \{\text{b}: \theta_3\}\gamma_{14}) \hookrightarrow (\{0: (\theta_3$ & (integer | truth-value))$\}, \{\ \})$;
$\varepsilon \vdash$ give the value stored in the cell bound to "b":
$(\{\ \}\gamma_{15}, \{\text{b}: \theta_4\}\gamma_{16}) \hookrightarrow (\{0: (\theta_5$ & (integer | truth-value))$\}, \{\ \})$

$\varepsilon \vdash$ ... or ... : $(\{\ \}\gamma_{17}, \{\text{b}: (\theta_3 \mid \theta_4)\}\gamma_{18}) \hookrightarrow$
$(\{0: (\theta_3$ & (integer | truth-value)) | $(\theta_5$ & (integer | truth-value))$\}, \{\ \})$

where U is extended with the substitution:

$$[\gamma_{13} \mapsto \{\ \}\gamma_{17}, \gamma_{15} \mapsto \{\ \}\gamma_{17}, \gamma_{14} \mapsto \{\ \}\gamma_{18}, \gamma_{16} \mapsto \{\ \}\gamma_{18}]$$

Eventually, application of rule (HENCE-I) will instantiate the sort variables θ_3 and θ_4 to true. Thus the antecedent θ_4 & cell[integer | truth-value] $\neq$ nothing is not satisfied, and the action on line 8 is ill-sorted. This action can be replaced by "fail", and the identity "A or fail = A" can be used to simplify the "or" action to "give the value bound to "b"".

A similar argument applies to the other "or" action, on lines 10–11. Because of the binding "x: cell[integer]", however, this "or" action is simplified to "give the value stored in the cell bound to "x"".

Finally, consider the sort inferred for the "unfolding" action on lines 6–15. To simplify the explanation of the sort inference, we will express the body of the "unfolding" as follows:

```
A  =  A1 then ((A2 and then unfold) else complete)

A1 =  give the value bound to "b" or
      give the value stored in the cell bound to "b"

A2 =  | give the value bound to "x" or
      | give the value stored in the cell bound to "x"
      then give negation (the integer)
      then store the value in the cell bound to "x"
```

From above, we already know the sort for action A_1 is:

$$\frac{\dots}{\varepsilon \vdash A_1\text{:}\quad (\{\ \}\gamma_{17}, \{\text{b: } (\theta_3 \mid \theta_4)\}\gamma_{18}) \hookrightarrow (\{0\text{: } (\theta_3 \ \& \ (\text{integer} \mid \text{truth-value})) \mid (\theta_5 \ \& \ (\text{integer} \mid \text{truth-value}))\}, \{\ \})}$$

and we can show that action A_2 has the following sort:

$$\frac{\ldots}{\varepsilon \vdash A_2\text{: }(\{\ \}\gamma_{19}, \{\mathsf{x}\text{: }(\theta_5 \mid \theta_6)\ \&\ \theta_7\}\gamma_{20}) \hookrightarrow (\{\ \}, \{\ \})}$$

(since A_2 contains three occurrences of the yielder "the S bound to "x"", sort inference will introduce three distinct θ-variables.)

For brevity, let $\sigma_1 = (\theta_3 \mid \theta_4)$ and $\sigma_2 = (\theta_5 \mid \theta_6)\ \&\ \theta_7$. Now, we can infer the sort of A as follows:

$$\varepsilon \vdash A_2\text{: }(\{\ \}\gamma_{19}, \{\mathsf{x}\text{: }\sigma_2\}\gamma_{20}) \hookrightarrow (\{\ \}, \{\ \})\ ;$$
$$\varepsilon \vdash \mathsf{unfold}\text{: }(\tau, \beta) \hookrightarrow (\tau', \beta')$$

$$\varepsilon \vdash A_2\ \mathsf{and\ then\ unfold}\text{: }(\mathit{distribute}\ \{\ \}\gamma_{19}\ \tau, \mathit{distribute}\ \{\mathsf{x}\text{: }\sigma_2\}\gamma_{20}\ \beta) \hookrightarrow$$
$$\qquad (\mathit{merge}\ \{\ \}\ \tau', \mathit{merge}\ \{\ \}\ \beta')\ ;$$
$$\varepsilon \vdash \mathsf{complete}\text{: }(\{\ \}\gamma_{21}, \{\ \}\gamma_{22}) \hookrightarrow (\{\ \}, \{\ \})\ ;$$
$$\mathit{distribute}\ \{\ \}\gamma_{19}\ \tau = \{\ \}\gamma_{21} = \{\mathsf{0}\text{: truth-value}\}\gamma_{23}\ ;$$
$$\mathit{merge}\ \{\ \}\ \tau' = \{\ \}$$

$$\varepsilon \vdash (A_2\ \mathsf{and\ then\ unfold})\ \mathsf{else\ complete}\text{:}$$
$$\qquad (\{\mathsf{0}\text{: truth-value}\}\gamma_{23}, \mathit{switch}\ (\mathit{distribute}\ \{\mathsf{x}\text{: }\sigma_2\}\gamma_{20}\ \beta)\ \{\ \}\gamma_{22}) \hookrightarrow$$
$$\qquad (\{\ \}, \mathit{select}\ (\mathit{merge}\ \{\ \}\ \beta')\ \{\ \})\ ;$$
$$\varepsilon \vdash A_1\text{: }\quad (\{\ \}\gamma_{17}, \{\mathsf{b}\text{: }\sigma_1\}\gamma_{18}) \hookrightarrow$$
$$\qquad (\{\mathsf{0}\text{: }(\theta_3\ \&\ (\text{integer} \mid \text{truth-value})) \mid (\theta_5\ \&\ (\text{integer} \mid \text{truth-value}))\}, \{\ \})$$

$$\varepsilon \vdash A\text{: }\quad (\{\ \}\gamma_{17}, \mathit{distribute}\ \{\mathsf{b}\text{: }\sigma_1\}\gamma_{18}\ (\mathit{switch}\ (\mathit{distribute}\ \{\mathsf{x}\text{: }\sigma_2\}\gamma_{20}\ \beta)\ \{\ \}\gamma_{22})) \hookrightarrow$$
$$\qquad (\{\ \}, \mathit{merge}\ \{\ \}\ (\mathit{select}\ (\mathit{merge}\ \{\ \}\ \beta')\ \{\ \}))$$

Since the action A must also have the sort $(\tau, \beta) \hookrightarrow (\tau', \beta')$, we can generate the following set of equations, and we use the properties of the auxiliary operations to simplify them:

$$\begin{aligned}
\tau &= \{\ \}\gamma_{17} \\
\beta &= \mathit{distribute}\ \{\mathsf{b}\text{: }\sigma_1\}\gamma_{18}\ (\mathit{switch}\ (\mathit{distribute}\ \{\mathsf{x}\text{: }\sigma_2\}\gamma_{20}\ \beta)\ \{\ \}\gamma_{22}) \\
&= \mathit{distribute}\ \{\mathsf{b}\text{: }\sigma_1\}\gamma_{18}\ (\mathit{distribute}\ \{\mathsf{x}\text{: }\sigma_2\}\gamma_{20}\ \beta))
\end{aligned}$$

$$
\begin{aligned}
&= \textit{distribute}\ (\textit{distribute}\ \{\mathsf{b}\colon \sigma_1\}\gamma_{18}\ \{\mathsf{x}\colon \sigma_2\}\gamma_{20}))\ \beta \\
&= \textit{distribute}\ \{\mathsf{b}\colon \sigma_1, \mathsf{x}\colon \sigma_2\}\gamma_{24}))\ \beta \\
&= \{\mathsf{b}\colon \sigma_1, \mathsf{x}\colon \sigma_2\}\gamma_{24} \\
\tau' &= \{\ \} \\
\beta' &= \textit{merge}\ \{\ \}\ (\textit{select}\ (\textit{merge}\ \{\ \}\ \beta')\ \{\ \}) \\
&= \textit{select}\ (\textit{merge}\ \{\ \}\ \beta')\ \{\ \} \\
&= \textit{select}\ \beta'\ \{\ \} \\
&= \{\ \}
\end{aligned}
$$

The final sorts assigned to the "unfolding A" and "unfold" actions are:

- unfolding ... : $(\{\ \}\gamma_{17}, \{\mathsf{b}\colon \sigma_1, \mathsf{x}\colon \sigma_2\}\gamma_{24}) \hookrightarrow (\{\ \}, \{\ \})$
- unfold: $(\{\ \}\gamma_{17}, \{\mathsf{b}\colon \sigma_1, \mathsf{x}\colon \sigma_2\}\gamma_{24}) \hookrightarrow (\{\ \}, \{\ \})$

6.5 Implementation of the Sort Inference Algorithm

The sort inference rules have been implemented to produce the action notation sort checker. The action notation sort checker accepts an action tree and performs sort inference on it. The result is an action tree where each node of the tree has been decorated with the sort of the action tree rooted at that node.

The notions of sort schemes and variables exist only within the sort checker. At the end of sort inference, all row variables, sort variables and field variables are instantiated, and all of the various schemes are eliminated, i.e. sort schemes are replaced by sorts, record sort schemes are replaced by record sorts, and field schemes are removed.

The implementation of our algorithm consists of three passes. The first pass annotates the given action with sort schemes, in accordance with the sort inference rules. The second pass reduces all sorts to canonical form, and removes all sort, field and row variables. The third pass marks places where run-time sort checks are required, replaces ill-sorted actions by "fail", simplifies the program action, and checks

Figure 6.8: Implementation of the (AND-THEN-I) sort rule

```
decorate_action (AND_THEN (a_1, a_2, _)) E =
    let
      val a_1' = decorate_action a_1 E
      val ((t_1, b_1), (t_1', b_1')) = get_action a_1'
      val a_2' = decorate_action a_2 E
      val ((t_2, b_2), (t_2', b_2')) = get_action a_2'
      val t = distribute t_1 t_2
      val b = distribute b_1 b_2
      val t' = merge t_1' t_2'
      val b' = merge b_1' b_2'
    in
      AND_THEN (a_1', a_2', Action((t,b), (t',b')))
    end
```

any constraints. We consider the three passes in more detail in the following sections.

6.5.1 Inferring the Sorts

The first pass in the action notation sort checker annotates the given action tree with record sort schemes. It consists of a collection of mutually-recursive SML functions that traverse the action tree and infer the sort of each node. The functions are classified according to the kind of term they expect, for example, an action, a yielder, or a data term. If one of these functions is applied to a node that is not of the expected kind, then it signals a sort error by raising an exception. Similarly, if any of the auxiliary operations return *failure,* then this is also treated as an exception.

For example, the function `decorate_action` is used to infer the sorts of any action terms. It consists of individual clauses that correspond to each of the sort inference rules for actions. Each clause in the function is a simple translation of the corresponding sort inference rule. For example, consider the implementation of the rule (AND-THEN-I) given in Figure 6.8. First, the function `decorate_action` is called recursively to infer the sort of the first sub-action `a_1`, using the same sort environment `E`. The sort of the resulting tree `a_1'` is then checked to make sure it is an action sort (`get_action a_1'`), and its input and output sorts are bound to the variables `t_1`, `b_1`, `t_1'` and `b_1'`. This process is then repeated for the second sub-action `a_2`. Next, the record sort schemes are combined using the appropriate auxiliary operations (`distribute` and `merge`). The generated substitution is stored in three global arrays, one for each type of variable (sort, field, and row), indexed by variable number. Finally, the decorated `AND_THEN` tree is constructed from the decorated trees for the sub-actions, and the action sort constructed from resulting record sort schemes.

In the first pass, any constraints occurring in a sort inference rule cannot be checked as there will not be sufficient information available. For example, sort variables are not instantiated until the inputs for one action are unified with the outputs from another, and this happens when the code of a rule for a node that is further up the action tree is executed, which is after the code of the rule that contained the constraint has been executed. Therefore, the checking of the constraints has to be delayed until later, and in fact takes place during the third pass.

6.5.2 Eliminating the Variables

Once the entire action tree has been decorated, the second pass traverses the tree and instantiates any remaining variables. If we assume that the action tree as a whole receives no transients or bindings, we can unify the input transients and bindings for the whole action with the empty record { }. Any remaining sort variables are instantiated to the sort datum, since this is the most general sort possible; any remaining field variables are instantiated to the field **absent**, since they must be ignored by the action; and any remaining row variables are instantiated to the empty record { }, again since any fields they represent must be ignored by the action.

Since all sort variables are now instantiated, all sort schemes can be replaced by ordinary sorts, and then these sorts can be reduced to their canonical form, i.e. all occurrences of sort meet can be eliminated, and sort joins can be simplified by removing redundant terms.

We also discard the instantiations of any γ-variables and remove any **absent** fields. This reduces the inferred record sort schemes to simple record sorts where the inputs sorts only include the fields required by the action. Some examples of the reduction of sort schemes to sorts are as follows:

- $\{0: \theta_1$ & (integer | truth-value)$\}$, where θ_1 is instantiated to integer, becomes $\{0:$ integer$\}$.
- $\{1: \theta_1\}\gamma_1$, where γ_1 is instantiated to $\{2:$ integer$\}$, becomes $\{1:$ datum$\}$.
- $\{$b: integer, x: **absent**$\}\gamma_2$ becomes $\{$b: integer$\}$.
- $\{$x: truth-value$\}\rho_3$, where ρ_3 is instantiated to $\{$b: integer$\}\gamma_4$, becomes $\{$b: integer, x: truth-value$\}$.

6.5.3 Simplifying the Action

The third pass is responsible for checking that all of the constraints have been satisfied, for detecting the places where run-time sort checks are needed, and for simplifying the program action by replacing ill-sorted sub-actions by "fail".

The action tree is traversed for the final time, and all sorts are checked to make sure they are not nothing. If an action or yielder sort contains any part that is nothing, then that action or yielder is ill-sorted. An ill-sorted yielder will cause the action that contains it to fail, and hence the action becomes ill-sorted. Any ill-sorted (sub-)action is replaced by "fail". Checking the action and yielder sorts for nothing detects most of the constraint violations, since most sorts that must be non-nothing also appear as part of the sort of an action or yielder. Any remaining constraints must be checked separately, and if they are not satisfied, then the node in the action tree which produced the constraint is ill-sorted. This will cause the enclosing sub-action to be re-written to "fail".

The third pass also determines if a run-time sort check will be necessary, and annotates the action tree accordingly. In general, if a yielder (e.g. "the *S*", "the *S* bound to *k*" or "the *S* stored in *Y*") expects a datum of sort *S*, and it has been inferred that the sort of the incoming datum is *S*′, then a run-time sort check will be necessary unless $S' \leq S$. For example, consider the following yielders and their sorts:

- the integer#1: ({1: 42}, { }) $\rightsquigarrow$ 42. This yielder receives a transient of sort 42 and expects it to be of sort integer. Since 42 ≤ integer, no run-time sort check is required.

- the integer bound to "n": ({ }, {n: integer I truth-value}) $\rightsquigarrow$ integer. This yielder receives a binding of sort (integer I truth-value) and expects it to be of sort integer. Since (integer I truth-value) $\not\leq$ integer, a run-time sort check is required to ensure that the received datum is of sort integer.

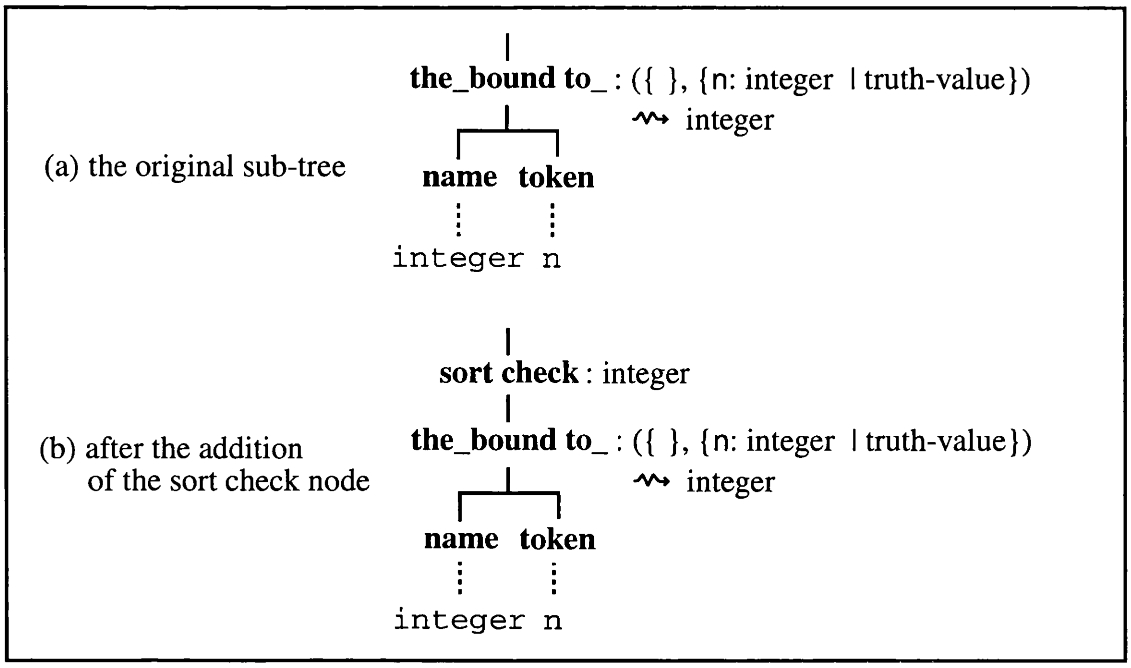
- the integer stored in the cell bound to "x": ({ }, {x: cell[integer I truth-value]}) $\rightsquigarrow$ integer. The yielder "the cell bound to "x"" does not require a run-time sort check, as the received datum is of sort cell. The datum stored in the cell is, however, of sort (integer I truth-value), and the whole yielder expects that this datum is of sort integer. Since (integer I truth-value) $\not\leq$ integer, a run-time sort check is required to ensure that the datum stored in a particular cell is of sort integer.

A run-time sort check is indicated in the action tree by introducing a new "sort check" node into the action tree, with the yielder as a sub-tree. The "sort check" node is decorated with the sort to be checked for. The insertion of the run-time sort check for the second example given above is shown in Figure 6.9.

Finally, the action tree is simplified using the algebraic laws for "fail". For example:

- *A* or fail = fail or *A* = *A*

Figure 6.9: An example of inserting a run-time sort check


(a) the original sub-tree
the_bound to_ : ({ }, {n: integer | truth-value}) ⇝ integer
name token
integer n
(b) after the addition of the sort check node
sort check : integer
the_bound to_ : ({ }, {n: integer | truth-value}) ⇝ integer
name token
integer n

- fail and then A = fail

As a result, it is possible that either the ill-sorted action can be removed (as a sub-action of "or"), or the entire action may be re-written to "fail". We expect the majority of "or" actions to be eliminated as a result of sort checking.

6.6 Conclusion

The action notation sort checker is a key component of the ACTRESS system. Without the sort information generated by this phase, action transformation and efficient code generation would be impossible. The action notation transformer relies solely on sort information to simplify the action tree. The action notation code generator relies on sort information to determine the transients and bindings required and produced by an action, and uses this knowledge to perform register allocation.

Our sort inference algorithm represents an extremely complex analysis of the functional and declarative facets of an action. It is capable of determining individual values used in an action, where there is sufficient information. Moreover, sort inference propagates these known values throughout the action to the places where they are used, and in doing so, performs a type of constant propagation similar to that found in a traditional compiler.

The sort inference algorithm relies on a collection of auxiliary operations to calculate the sort of a composite action by combining the sorts of its sub-actions. These auxiliary operations allow us to formalise the sort inference algorithm concisely, building on the similarities between different action combinators, but clearly showing their differences.

We have specified our sort inference algorithm for ACTRESS action notation using a collection of sort inference rules, and implemented the algorithm using a reasonably systematic translation of the inference rules into SML code. Whilst not a formal proof of the correctness of the action notation sort checker, this correspondence between an inference rule and its implementation reduces the likelihood of errors in the sort checker.

Finally, we believe the action notation sort checker represents the most sophisticated analysis of actions to date. Some other systems, e.g. CANTOR[Pal92b], only handle actions that do not require run-time sort checks. Our ability to accept actions that do require run-time sort checks, and to annotate the places where such checks are required, is unique. Some systems, e.g. CANTOR, perform sort inference in a strictly bottom-up fashion. Our more general method is essential to handle the subset of action notation used in ACTRESS.

Chapter 7

Soundness of the Sort Inference Algorithm

7.1 Introduction

In this chapter, we are concerned with the *soundness* of the sort inference rules with respect to the semantics of action notation. Soundness proves that the sort we infer for an action is consistent with the transients and bindings received and produced by the action, when it is performed. We prove soundness by relating each sort inference rule given in Appendix B to its corresponding semantic rules given in Appendix A.

Before we can present the proof of soundness, however, we must formalise some properties of the auxiliary operations as a number of lemmas. These lemmas are structured in a hierarchy—the lemma for an auxiliary operation uses the lemmas for its component field and row operations. Section 7.2 gives some definitions required for the proofs; Section 7.3 presents the lemmas for each of the sort, field and row operations used in the auxiliary operations; Section 7.4 proves the ordering properties of the auxiliary operations given in Section 6.2.6; and Section 7.5 presents lemmas for the auxiliary operations needed in the proof of soundness. Next, Section 7.6 formalises the soundness property for ACTRESS action notation, and Section 7.7 presents its proof. Finally, Section 7.8 concludes and briefy discusses the completeness of the sort inference algorithm.

7.2 Definitions

Definition: a *substitution* U is a triple (U_r, U_f, U_s), where each U_i is a total mapping from variables to schemes. The U_i's differ only in the kinds of variables and schemes: U_r maps row variables (ρ_i and γ_i) to record sort schemes; U_f maps field variables (Δ_i) to field schemes; and U_s maps sort variables (θ_i) to sort schemes. A substitution U can be applied to a sort scheme, σ, τ, or β, to replace all of the variables mapped by U that occur in the scheme with their corresponding instantiations, denoted $U(\sigma)$, $U(\tau)$, or $U(\beta)$ respectively, in the normal way. Finally, if applying a substitution U maps a scheme to a *ground scheme* (i.e. one with no variables), then U is called a *ground substitution* of that scheme.

Definition: a datum d is an *instance* of a sort S, written $d \in S$, if $d : S$, i.e. d is an individual of sort S. If d is not an instance of S, we write $d \notin S$.

Definition: a datum d is an instance of a sort scheme σ, if there exists a ground substitution U such that $d \in U(\sigma)$.

Definition: a map of transients t is an *instance* of a ground transient sort scheme τ if $t \in \tau$, i.e. *dom* $t \supseteq$ *dom* τ, and $t(i) \in \tau(i)$, for all $i \in$ *dom* τ. Similarly, a map of bindings b is an instance of a ground binding sort scheme β if $b \in \beta$.

Definition: a map of transients t is an *instance* of a transient sort scheme τ if there exists a ground substitution U such that $t \in U(\tau)$, i.e. *dom* $t \supseteq$ *dom* $U(\tau)$, and $t(i) \in U(\tau)(i)$, for all $i \in$ *dom* $U(\tau)$. Similarly, a map of bindings b is an instance of a binding sort scheme β if $\exists U$ s.t. $b \in U(\beta)$.

Definition: a transient sort scheme τ is a *subsort* of a transient sort scheme τ', written $\tau \sqsubseteq \tau'$, if and only if for all ground substitutions U, *dom* $U(\tau) \supseteq$ *dom* $U(\tau')$ and $U(\tau)(i) \leq U(\tau')(i)$, for all $i \in$ *dom* $U(\tau')$. Similarly for binding sort schemes β and β' ($\beta \sqsubseteq \beta'$).

7.3 Lemmas

7.3.1 Lemma for the Sort Operation

Lemma 1: If $U, \sigma = \mathit{distribute}_{\mathit{sort}}\, \sigma_i\, \sigma_j$ then $\sigma = U(\sigma_i \,\&\, \sigma_j)$.

Proof: The proof is constructed inductively over the structure of sort schemes by considering each equation of the $\mathit{distribute}_{\mathit{sort}}$ operation in turn.

Case 1: $\sigma_i = \theta_i,\ \sigma_j = \theta_j$

$U = [\theta_i \mapsto \theta_m,\ \theta_j \mapsto \theta_m],\ \sigma = \theta_m$

$U(\sigma_i \,\&\, \sigma_j) \quad = U(\theta_i \,\&\, \theta_j) \quad = \theta_m \,\&\, \theta_m \quad = \theta_m$

Case 2: $\sigma_i = \theta,\ \sigma_j = \sigma_k$, where $\sigma_k \neq \theta$

$U = [\theta \mapsto \sigma_k],\ \sigma = \sigma_k$

$U(\sigma_i \,\&\, \sigma_j) \quad = U(\theta \,\&\, \sigma_k) \quad = \sigma_k \,\&\, \sigma_k \quad = \sigma_k$

Case 3: $\sigma_i = C[\sigma_k],\ \sigma_j = C[\sigma_l]$

$\sigma = C[\sigma']$ where $U, \sigma' = \mathit{distribute}_{\mathit{sort}}\, \sigma_k\, \sigma_l$

$U(\sigma_i \,\&\, \sigma_j) \quad = U(C[\sigma_k] \,\&\, C[\sigma_l])$

$= U(C[\sigma_k \,\&\, \sigma_l])$

$= C[U(\sigma_k \,\&\, \sigma_l)]$

$= C[\sigma']$, by induction

since $\sigma' = U(\sigma_k \,\&\, \sigma_l)$

Case 4: $\sigma_i = I,\ \sigma_j = B$, where $I \in B$

$U = [\,],\ \sigma = I$

$U(\sigma_i \,\&\, \sigma_j) \quad = U(I \,\&\, B) \quad = I \,\&\, B \quad = I$

Case 5: $\sigma_i = I,\ \sigma_j = I$

$U = [\,],\ \sigma = I$

$U(\sigma_i \,\&\, \sigma_j) \quad = U(I \,\&\, I) \quad = I \,\&\, I \quad = I$

Case 6: $\sigma_i = B$, $\sigma_j = B$

$U = [\,]$, $\sigma = B$

$U(\sigma_i \mathbin{\&} \sigma_j) \quad = U(B \mathbin{\&} B) \quad = B \mathbin{\&} B \quad = B$

Case 7: $\sigma_i = \text{nothing}$, $\sigma_j = \sigma_k$

$U = [\,]$, $\sigma = \text{nothing}$

$U(\sigma_i \mathbin{\&} \sigma_j) \quad = U(\text{nothing} \mathbin{\&} \sigma_k) \quad = \text{nothing} \mathbin{\&} \sigma_k \quad = \text{nothing}$

Case 8: $\sigma_i = \text{datum}$, $\sigma_j = \sigma_k$

$U = [\,]$, $\sigma = \sigma_k$

$U(\sigma_i \mathbin{\&} \sigma_j) \quad = U(\text{datum} \mathbin{\&} \sigma_k) \quad = \text{datum} \mathbin{\&} \sigma_k \quad = \sigma_k$

Case 9: $\sigma_i = (\sigma_k \mid \sigma_l)$, $\sigma_j = \sigma_m$

$U = [\,]$, $\sigma = (\sigma_k \mid \sigma_l) \mathbin{\&} \sigma_m$

$U(\sigma_i \mathbin{\&} \sigma_j) \quad = U((\sigma_k \mid \sigma_l) \mathbin{\&} \sigma_m) \quad = (\sigma_k \mid \sigma_l) \mathbin{\&} \sigma_m$

Case 10: $\sigma_i = (\sigma_k \mathbin{\&} \sigma_l)$, $\sigma_j = \sigma_m$

$U = U_l \circ U_k$, $\sigma = U_l(\sigma_k') \mathbin{\&} \sigma_l'$ where $U_k, \sigma_k' = \mathit{distribute}_{sort}\ \sigma_k\ \sigma_m$

and $U_l, \sigma_l' = \mathit{distribute}_{sort}\ U_k(\sigma_l)\ U_k(\sigma_m)$

$$\begin{aligned}
U(\sigma_i \mathbin{\&} \sigma_j) &= U((\sigma_k \mathbin{\&} \sigma_l) \mathbin{\&} \sigma_m)\\
&= U((\sigma_k \mathbin{\&} \sigma_m) \mathbin{\&} (\sigma_l \mathbin{\&} \sigma_m))\\
&= U(\sigma_k \mathbin{\&} \sigma_m) \mathbin{\&} U(\sigma_l \mathbin{\&} \sigma_m)\\
&= (U_l \circ U_k)(\sigma_k \mathbin{\&} \sigma_m) \mathbin{\&} (U_l \circ U_k)(\sigma_l \mathbin{\&} \sigma_m)\\
&= U_l(U_k(\sigma_k \mathbin{\&} \sigma_m)) \mathbin{\&} U_l(U_k(\sigma_l \mathbin{\&} \sigma_m))\\
&= U_l(\sigma_k') \mathbin{\&} \sigma_l'
\end{aligned}$$

since $\sigma_k' = U_k(\sigma_k \mathbin{\&} \sigma_m)$ and $\sigma_l' = U_l(U_k(\sigma_l) \mathbin{\&} U_k(\sigma_m)) = U_l(U_k(\sigma_l \mathbin{\&} \sigma_m))$

7.3.2 Lemmas for the Field Operations

Lemma 2: If U, $\phi = \mathit{distribute}_{field}\ \phi_i\ \phi_j$ then

(1) $\phi = \sigma$ iff $\sigma \leq U(\phi_i)$ and $\sigma \leq U(\phi_j)$.

(2) $\phi = \textbf{absent}$ iff $U(\phi_i) = U(\phi_j) = \textbf{absent}$.

(3) $\phi = \Delta$ iff $U(\phi_i) = U(\phi_j) = \Delta$.

Proof: For each result ϕ, we consider the possible kinds of inputs ϕ_i and ϕ_j which can produce that kind of result given the definition of $distribute_{field}$.

Case 1: $\phi = \sigma$

Subcase 1: $\phi_i = \sigma_i$, $\phi_j = \sigma_j$

U, $\phi = distribute_{sort}\, \sigma_i\, \sigma_j$, by Lemma 1, $\phi = \sigma = U(\sigma_i \,\&\, \sigma_j) = U(\phi_i) \,\&\, U(\phi_j)$.

Therefore, $\sigma \leq U(\phi_i)$ and $\sigma \leq U(\phi_j)$ by the properties of sort meet.

Subcase 2: $\phi_i = \Delta$, $\phi_j = \sigma$

$U = [\Delta \mapsto \sigma]$, $\phi = \sigma$.

$U(\phi_i) = U(\Delta) = \sigma$ $\qquad$ $U(\phi_j) = U(\sigma) = \sigma$

Subcase 3: $\phi_i = \sigma$, $\phi_j = \Delta$

Follows from *Subcase 2*, since $distribute_{field}$ is commutative.

Therefore, $\phi = \sigma$ implies $\sigma \leq U(\phi_i)$ and $\sigma \leq U(\phi_j)$

Case 2: $\phi =$ **absent**

Subcase 1: $\phi_i =$ **absent**, $\phi_j =$ **absent**

$U = [\ ]$, $\phi =$ **absent**

$U(\phi_i) =$ **absent** $\qquad$ $U(\phi_j) =$ **absent**

Subcase 2: $\phi_i = \Delta$, $\phi_j =$ **absent**

$U = [\Delta \mapsto$ **absent**$]$, $\phi =$ **absent**

$U(\phi_i) = U(\Delta) =$ **absent** $\qquad$ $U(\phi_j) =$ **absent**

Subcase 3: $\phi_i =$ **absent**, $\phi_j = \Delta$

Follows from *Subcase 2*, since $distribute_{field}$ is commutative.

Therefore, $\phi =$ **absent** implies $U(\phi_i) = U(\phi_j) =$ **absent**.

Case 3: $\phi = \Delta$

Subcase 1: $\phi_i = \Delta_i$, $\phi_j = \Delta_j$, where $\Delta_i = \Delta_j$

$U = [\ ]$, $\phi = \Delta_i$

$U(\phi_i) = \Delta_i$ $\qquad$ $U(\phi_j) = \Delta_i$

Subcase 2: $\phi_i = \Delta_i$, $\phi_j = \Delta_j$, where $\Delta_i \neq \Delta_j$

$U = [\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta]$, $\phi = \Delta$

$U(\phi_i) = U(\Delta_i) = \Delta$ $\qquad$ $U(\phi_j) = U(\Delta_j) = \Delta$

Therefore, $\phi = \Delta$ implies $U(\phi_i) = U(\phi_j) = \Delta$.

Lemma 3: If U, $\phi = merge_{field}\ \phi_i\ \phi_j$ then

(1) $\phi = \sigma$ implies $U(\phi_i) =$ **absent** and $U(\phi_j) = \sigma$ or $U(\phi_i) = \sigma$ and $U(\phi_j) =$ **absent**.

(2) $\phi =$ **absent** implies $U(\phi_i) = U(\phi_j) =$ **absent**.

(3) $\phi = \Delta$ implies $U(\phi_i) =$ **absent** and $U(\phi_j) = \Delta$ or $U(\phi_i) = \Delta$ and $U(\phi_j) =$ **absent**.

Proof: For each result ϕ, we consider the possible kinds of inputs ϕ_i and ϕ_j which can produce that kind of result given the definition of $merge_{field}$.

Case 1: $\phi = \sigma$

Subcase 1: $\phi_i = \sigma$, $\phi_j =$ **absent**

$U = [\ ]$, $\phi = \sigma$

$U(\phi_i) = \sigma$ $\qquad$ $U(\phi_j) =$ **absent**

Subcase 2: $\phi_i = \sigma$, $\phi_j = \Delta$

$U = [\Delta \mapsto$ **absent**$]$, $\phi = \sigma$

$U(\phi_i) = \sigma$ $\qquad$ $U(\phi_j) = U(\Delta) =$ **absent**

Subcase 3: $\phi_i =$ **absent**, $\phi_j = \sigma$

$U = [\,], \phi = \sigma$

$U(\phi_i) = \textbf{absent}$ $\quad$ $U(\phi_j) = \sigma$

Subcase 4: $\phi_i = \Delta, \phi_j = \sigma$

$U = [\Delta \mapsto \textbf{absent}], \phi = \sigma$

$U(\phi_i) = U(\Delta) = \textbf{absent}$ $\quad$ $U(\phi_j) = \sigma$

Case 2: $\phi = \textbf{absent}$

Subcase 1: $\phi_i = \textbf{absent}, \phi_j = \textbf{absent}$

$U = [\,], \phi = \textbf{absent}$

$U(\phi_i) = \textbf{absent}$ $\quad$ $U(\phi_j) = \textbf{absent}$

Subcase 2: $\phi_i = \Delta, \phi_j = \Delta$

$U = [\Delta \mapsto \textbf{absent}], \phi = \textbf{absent}$

$U(\phi_i) = U(\Delta) = \textbf{absent}$ $\quad$ $U(\phi_j) = U(\Delta) = \textbf{absent}$

Case 3: $\phi = \Delta$

Subcase 1: $\phi_i = \Delta, \phi_j = \textbf{absent}$

$U = [\,], \phi = \Delta$

$U(\phi_i) = \Delta$ $\quad$ $U(\phi_j) = \textbf{absent}$

Subcase 2: $\phi_i = \textbf{absent}, \phi_j = \Delta$

$U = [\,], \phi = \Delta$

$U(\phi_i) = \textbf{absent}$ $\quad$ $U(\phi_j) = \Delta$

Lemma 4: If $U, \phi = switch_{field}\ \phi_i\ \phi_j$ then

(1) $\phi = \sigma$ implies $U(\phi_i) \leq \sigma$ and $U(\phi_j) \leq \sigma$.

(2) $\phi = \textbf{absent}$ implies $U(\phi_i) = U(\phi_j) = \textbf{absent}$.

(3) $\phi = \Delta$ implies $U(\phi_i) = U(\phi_j) = \Delta$.

Proof: For each result ϕ, we consider the possible kinds of inputs ϕ_i and ϕ_j which can produce that kind of result given the definition of $switch_{field}$.

Case 1: $\phi = \sigma$

Subcase 1: $\phi_i = \sigma_i$, $\phi_j = \sigma_j$

$U = [\,]$, $\phi = \sigma_i \mid \sigma_j$. Therefore, $U(\phi_i) \leq \sigma$ and $U(\phi_j) \leq \sigma$ by the properties of sort join.

Subcase 2: $\phi_i = \Delta$, $\phi_j = \sigma$

$U = [\Delta \mapsto \sigma]$, $\phi = \sigma$.

$U(\phi_i) = U(\Delta) = \sigma$ $\qquad$ $U(\phi_j) = U(\sigma) = \sigma$

Subcase 3: $\phi_i = \sigma$, $\phi_j = \Delta$

Follows from *Subcase 2*, since $switch_{field}$ is commutative.

Therefore, $\phi = \sigma$ implies $U(\phi_i) \leq \sigma$ and $U(\phi_j) \leq \sigma$.

Case 2: $\phi =$ **absent**

Subcase 1: $\phi_i =$ **absent**, $\phi_j =$ **absent**

$U = [\,]$, $\phi =$ **absent**

$U(\phi_i) =$ **absent** $\qquad$ $U(\phi_j) =$ **absent**

Subcase 2: $\phi_i = \Delta$, $\phi_j =$ **absent**

$U = [\Delta \mapsto$ **absent**$]$, $\phi =$ **absent**

$U(\phi_i) = U(\Delta) =$ **absent** $\qquad$ $U(\phi_j) =$ **absent**

Subcase 3: $\phi_i =$ **absent**, $\phi_j = \Delta$

Follows from *Subcase 2*, since $switch_{field}$ is commutative.

Therefore, $\phi =$ **absent** implies $U(\phi_i) = U(\phi_j) =$ **absent**.

Case 3: $\phi = \Delta$

Subcase 1: $\phi_i = \Delta_i$, $\phi_j = \Delta_j$, where $\Delta_i = \Delta_j$

$U = [\,]$, $\phi = \Delta_i$

$U(\phi_i) = \Delta_i$ $\qquad\qquad$ $U(\phi_j) = \Delta_i$

Subcase 2: $\phi_i = \Delta_i$, $\phi_j = \Delta_j$, where $\Delta_i \neq \Delta_j$

$U = [\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta]$, $\phi = \Delta$

$U(\phi_i) = U(\Delta_i) = \Delta$ $\qquad\qquad$ $U(\phi_j) = U(\Delta_j) = \Delta$

Therefore, $\phi = \Delta$ implies $U(\phi_i) = U(\phi_j) = \Delta$.

Lemma 5: If U, $\phi = select_{field}\ \phi_i\ \phi_j$ then

(1) $\phi = \sigma$ iff $U(\phi_i) \leq \sigma$ and $U(\phi_j) \leq \sigma$.

(2) $\phi =$ **absent** iff $U(\phi_i) = U(\phi_j) =$ **absent**.

(3) $\phi = \Delta$ iff $U(\phi_i) = U(\phi_j) = \Delta$.

Proof: For each result ϕ, we consider the possible kinds of inputs ϕ_i and ϕ_j which can produce that kind of result given the definition of $select_{field}$.

Case 1: $\phi = \sigma$

Subcase 1: $\phi_i = \sigma_i$, $\phi_j = \sigma_j$

$U = [\,]$, $\phi = \sigma_i \mid \sigma_j$. Therefore, $U(\phi_i) \leq \sigma$ and $U(\phi_j) \leq \sigma$ by the properties of sort join.

Case 2: $\phi =$ **absent**

Subcase 1: $\phi_i =$ **absent**, $\phi_j =$ **absent**

$U = [\,]$, $\phi =$ **absent**

$U(\phi_i) =$ **absent** $\qquad\qquad$ $U(\phi_j) =$ **absent**

Case 3: $\phi = \Delta$

Subcase 1: $\phi_i = \Delta_i$, $\phi_j = \Delta_j$, where $\Delta_i = \Delta_j$

$U = [\,], \phi = \Delta_i$

$U(\phi_i) = \Delta_i$ $\qquad$ $U(\phi_j) = \Delta_i$

Subcase 2: $\phi_i = \Delta_i, \phi_j = \Delta_j$, where $\Delta_i \neq \Delta_j$

$U = [\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta], \phi = \Delta$

$U(\phi_i) = U(\Delta_i) = \Delta$ $\qquad$ $U(\phi_j) = U(\Delta_j) = \Delta$

Lemma 6: If U, $\phi = \mathit{overlay}_{\mathit{field}}\ \phi_i\ \phi_j$ then

(1) $\phi' = \sigma$ iff $U(\phi_i) = \sigma$ or $U(\phi_i) =$ **absent** and $U(\phi_j) = \sigma$.

(2) $\phi =$ **absent** iff $U(\phi_i) = U(\phi_j) =$ **absent**.

(3) $\phi = \Delta$ iff $U(\phi_i) = \Delta$ or $U(\phi_i) =$ **absent** and $U(\phi_j) = \Delta$.

Proof: For each result ϕ, we consider the possible kinds of inputs ϕ_i and ϕ_j which can produce that kind of resu $\mathit{overlay}_{\mathit{field}}$.

Subcase 3: $\phi_i = \sigma_i, \phi_j = \Delta_j$

$U = [\,], \phi = \sigma_i$

$U(\phi_i) = \sigma_i$

Case 1: $\phi = \sigma$

Subcase 1: $\phi_i = \sigma_i, \phi_j = \sigma_j$

$U = [\,], \phi = \sigma_i$

$U(\phi_i) = \sigma_i$ $\qquad$ $U(\phi_j) = \sigma_j$

Subcase 2: $\phi_i = \sigma_i, \phi_j =$ **absent**

Subcase 2: $\phi_i = \sigma_i, \phi_j =$ **absent**

U

$U = [\,], \phi = \sigma_i$

U

$U(\phi_i) = \sigma_i$ $\qquad$ $U(\phi_j) =$ **absent**

Subcase 3: $\phi_i = \sigma_i, \phi_j = \Delta_j$

$U = [\,], \phi = \sigma_i$

$U(\phi_i) = \sigma_i$ $\qquad$ $U(\phi_j) = \Delta_j$

Subcase 4: $\phi_i =$ **absent**, $\phi_j = \sigma_j$

$U = [\,], \phi = \sigma_i$

$U(\phi_i) =$ **absent** $\qquad$ $U(\phi_j) = \sigma_j$

Case 2: ϕ = **absent**

Subcase 1: ϕ_i = **absent**, ϕ_j = **absent**

$U = [\,]$, ϕ = **absent**

$U(\phi_i)$ = **absent** $\qquad$ $U(\phi_j)$ = **absent**

Case 3: $\phi = \Delta$

Subcase 1: ϕ_i = **absent**, $\phi_j = \Delta_j$

$U = [\,]$, $\phi = \Delta_j$

$U(\phi_i)$ = **absent** $\qquad$ $U(\phi_j) = \Delta_j$

Subcase 2: $\phi_i = \Delta$, $\phi_j = \Delta$

$U = [\,]$, $\phi = \Delta$

$U(\phi_i) = \Delta$ $\qquad$ $U(\phi_j) = \Delta$

7.3.3 Lemmas for the Row Operations

In the following lemmas, we use the variable $\Re$ to range over the kinds of row component of a record sort scheme, namely **exactly**, ρ, and γ.

Lemma 7: If U, $\Re = distribute_{row}\ \Re_i\ \Re_j$ then $\Re = U(\Re_i) = U(\Re_j)$.

Proof: For each result $\Re$, we consider the possible kinds of inputs $\Re_i$ and $\Re_j$ which can produce that kind of result given the definition of $distribute_{row}$.

Case 1: $\Re$ = **exactly**

Subcase 1: $\Re_i$ = **exactly**, $\Re_j$ = **exactly**

$U = [\,]$, $\Re = \rho_i$

$U(\Re_i)$ = **exactly** $\qquad$ $U(\Re_j)$ = **exactly**

Subcase 2: $\Re_i$ = **exactly**, $\Re_j = \rho$

$U = [\rho \mapsto$ **exactly**$]$, $\Re$ = **exactly**

$U(\Re_i) = \textbf{exactly}$ $U(\Re_j) = U(\rho) = \textbf{exactly}$

Subcase 3: $\Re_i = \rho$, $\Re_j = \textbf{exactly}$

Follows from *Subcase 2*, since $\textit{distribute}_{row}$ is commutative.

Subcase 4: $\Re_i = \textbf{exactly}$, $\Re_j = \gamma$

$U = [\gamma \mapsto \textbf{exactly}]$, $\Re = \textbf{exactly}$

$U(\Re_i) = \textbf{exactly}$ $U(\Re_j) = U(\gamma) = \textbf{exactly}$

Subcase 5: $\Re_i = \gamma$, $\Re_j = \textbf{exactly}$

Follows from *Subcase 4*, since $\textit{distribute}_{row}$ is commutative.

Case 2: $\Re = \rho$

Subcase 1: $\Re_i = \rho_i$, $\Re_j = \rho_j$ where $\rho_i = \rho_j$

$U = [\,]$, $\Re = \rho_i$

$U(\Re_i) = \rho_i$ $U(\Re_j) = \rho_i$

Subcase 2: $\Re_i = \rho_i$, $\Re_j = \rho_j$ where $\rho_i \neq \rho_j$

$U = [\rho_i \mapsto \rho, \rho_j \mapsto \rho]$, $\Re = \rho$

$U(\Re_i) = U(\rho_i) = \rho$ $U(\Re_j) = U(\rho_j) = \rho$

Subcase 3: $\Re_i = \rho_i$, $\Re_j = \gamma_j$

$U = [\gamma_j \mapsto \rho_i]$, $\Re = \rho_i$

$U(\Re_i) = U(\rho_i) = \rho_i$ $U(\Re_j) = U(\gamma_j) = \rho_i$

Subcase 4: $\Re_i = \gamma_i$, $\Re_j = \rho_j$

Follows from *Subcase 3*, since $\textit{distribute}_{row}$ is commutative.

Case 3: $\Re = \gamma$

Subcase 1: $\Re_i = \gamma_i$, $\Re_j = \gamma_j$ where $\gamma_i = \gamma_j$

$U = [\,]$, $\Re = \gamma_i$

$U(\mathfrak{R}_i) = \gamma_i$ $\qquad\qquad$ $U(\mathfrak{R}_j) = \gamma_i$

Subcase 2: $\mathfrak{R}_i = \gamma_i, \mathfrak{R}_j = \gamma_j$ where $\gamma_i \neq \gamma_j$

$U = [\gamma_i \mapsto \gamma, \gamma_j \mapsto \gamma], \mathfrak{R} = \gamma$

$U(\mathfrak{R}_i) = U(\gamma_i) = \gamma$ $\qquad\qquad$ $U(\mathfrak{R}_j) = U(\gamma_j) = \gamma$

Lemma 8: If U, $\mathfrak{R} = merge_{row}\ \mathfrak{R}_i\ \mathfrak{R}_j$ then

(1) $\mathfrak{R}$ = **exactly** iff $U(\mathfrak{R}_i) = U(\mathfrak{R}_j)$ = **exactly**.

(2) $\mathfrak{R} = \rho$ iff ($U(\mathfrak{R}_i) = \rho$ or $U(\mathfrak{R}_j) = \rho$) and $U(\mathfrak{R}_i) \neq U(\mathfrak{R}_j)$.

Proof: For each result $\mathfrak{R}$, we consider the possible kinds of inputs $\mathfrak{R}_i$ and $\mathfrak{R}_j$ which can produce that kind of result given the definition of $merge_{row}$.

Case 1: $\mathfrak{R}$ = **exactly**

Subcase 1: $\mathfrak{R}_i$ = **exactly**, $\mathfrak{R}_j$ = **exactly**

$U = [\,]$, $\mathfrak{R}$ = **exactly**

$U(\mathfrak{R}_i)$ = **exactly** $\qquad\qquad$ $U(\mathfrak{R}_j)$ = **exactly**

Subcase 2: $\mathfrak{R}_i = \rho, \mathfrak{R}_j = \rho$

$U = [\rho \mapsto$ **exactly**$]$, $\mathfrak{R}$ = **exactly**

$U(\mathfrak{R}_i) = U(\rho)$ = **exactly** $\qquad\qquad$ $U(\mathfrak{R}_j) = U(\rho)$ = **exactly**

Case 2: $\mathfrak{R} = \rho$

Subcase 1: $\mathfrak{R}_i$ = **exactly**, $\mathfrak{R}_j = \rho$

$U = [\,], \mathfrak{R} = \rho$

$U(\mathfrak{R}_i)$ = **exactly** $\qquad\qquad$ $U(\mathfrak{R}_j) = \rho$

Subcase 2: $\mathfrak{R}_i = \rho$, $\mathfrak{R}_j$ = **exactly**

$U = [\,], \mathfrak{R} = \rho$

$U(\mathfrak{R}_i) = \rho$ $\qquad\qquad$ $U(\mathfrak{R}_j)$ = **exactly**

Lemma 9: If U, $\mathfrak{R} = overlay_{row}\ \mathfrak{R}_i\ \mathfrak{R}_j$ then

(1) $\mathfrak{R} =$ **exactly** iff $U(\mathfrak{R}_i) = U(\mathfrak{R}_j) =$ **exactly**.

(2) $\mathfrak{R} = \rho$ iff $U(\mathfrak{R}_i) = U(\mathfrak{R}_j) = \rho$ or $U(\mathfrak{R}_i) =$ **exactly** and $U(\mathfrak{R}_j) = \rho$.

Proof: For each result $\mathfrak{R}$, we consider the possible kinds of inputs $\mathfrak{R}_i$ and $\mathfrak{R}_j$ which can produce that kind of result given the definition of $overlay_{row}$.

Case 1: $\mathfrak{R} =$ **exactly**

Subcase 1: $\mathfrak{R}_i =$ **exactly**, $\mathfrak{R}_j =$ **exactly**

$U = [\,]$, $\mathfrak{R} = \rho_i$

$U(\mathfrak{R}_i) =$ **exactly** $\qquad U(\mathfrak{R}_j) =$ **exactly**

Case 2: $\mathfrak{R} = \rho$

Subcase 1: $\mathfrak{R}_i =$ **exactly**, $\mathfrak{R}_j = \rho$

$U = [\,]$, $\mathfrak{R} = \rho$

$U(\mathfrak{R}_i) =$ **exactly** $\qquad U(\mathfrak{R}_j) = \rho$

Subcase 2: $\mathfrak{R}_i = \rho$, $\mathfrak{R}_j = \rho$

$U = [\,]$, $\mathfrak{R} = \rho$

$U(\mathfrak{R}_i) = \rho \qquad U(\mathfrak{R}_j) = \rho$

7.4 Ordering of the Auxiliary Operations

Recall that in Section 6.2.6 we asserted that the auxiliary operations obeyed the following ordering relationships:

- *distribute* $\tau_1\ \tau_2 \sqsubseteq \tau_1$
- *merge* $\tau_1\ \tau_2 \sqsubseteq \tau_1$
- $\tau_1 \sqsubseteq$ *select* $\tau_1\ \tau_2$

Since *distribute*, *merge* and *select* generate an implicit substitution, as well as a record sort scheme, we must re-state these subsort relationships in terms of the equivalent $distribute_p$, $merge_p$ and $select_p$ operations which make the substitution explicit. Thus the properties become:

- $\tau' \sqsubseteq U'(\tau_1)$, where $U', \tau' = distribute_p\ \tau_1\ \tau_2$
- $\tau' \sqsubseteq U'(\tau_1)$, where $U', \tau' = merge_p\ \tau_1\ \tau_2$
- $U'(\tau_1) \sqsubseteq \tau'$, where $U', \tau' = select_p\ \tau_1\ \tau_2$

We prove each of these properties in the following sections.

7.4.1 Ordering of distribute

Lemma 10: $\tau' \sqsubseteq U'(\tau_1)$, where $U', \tau' = distribute_p\ \tau_1\ \tau_2$

In order to show that $\tau' \sqsubseteq U'(\tau_1)$, we must show that for all ground substitutions U, $dom\ U(\tau') \supseteq dom\ (U \circ U')(\tau_1)$ and $U(\tau')(i) \leq (U \circ U')(\tau_1)(i)$, for all $i \in dom\ (U \circ U')(\tau_1)$, where $U', \tau' = distribute_p\ \tau_1\ \tau_2$.

If $i \in dom\ U(\tau')$ then there are three ways that the i-field could have arisen:

(1) τ' contains a field "i: σ_i", i.e. the i-field is present in the original record sort scheme (and the substitution U instantiates any sort variables in σ_i).

(2) τ' contains a field "i: Δ_i" and U contains the mapping "$\Delta_i \mapsto \sigma_i$", i.e. the i-field is bound to a field variable in the original record sort scheme and the substitution U instantiates it to a present field.

(3) τ' contains a row variable ρ (or γ) and U contains the mapping "$\rho \mapsto \{\ldots, i: \sigma_i, \ldots\}$" (or "$\gamma \mapsto \{\ldots, i: \sigma_i, \ldots\}$"), i.e. the i-field is not in the original record sort scheme but the substitution U instantiates a row variable to include it.

In practice, the substitution U may not instantiate a variable in τ' to a ground term immediately, i.e. it may instantiate a variable to a term still involving other variables, these are then instantiated to ground terms. Such a substitution, however, can always be re-written to one which does instantiate all variables immediately to ground terms. Therefore, we only need to consider the above three cases in order to prove the ordering property.

Proof: Consider each of these three cases in turn:

Case 1: $\tau'(i) = \sigma_i$

By Lemma 2(1), since $\tau'(i) = \sigma_i$, we have $\sigma_i \leq U'(\tau_1)(i)$, i.e. $U'(\tau_1)$ contains a field "i: $\sigma_i^?$" where $\sigma_i \leq \sigma_i'$. I.e. if $i \in$ *dom* $(U \circ U')(\tau_1)$ then $i \in$ *dom* $U(\tau')$ and $U(\tau')(i) \leq (U \circ U')(\tau_1)(i)$.

Case 2: $\tau'(i) = \Delta_i$ and $U(\Delta_i) = \sigma_i$

By Lemma 2(3), since $\tau'(i) = \Delta_i$, we have $U'(\tau_1)(i) = \Delta_i$. Therefore, if $U(\tau')(i) = \sigma_i$ then $(U \circ U')(\tau_1)(i) = \sigma_i$. I.e. if $i \in$ *dom* $U(\tau')$ then $i \in$ *dom* $(U \circ U')(\tau_1)$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i)$.

Case 3: *row* $\tau' = \rho$ and $U(\rho)(i) = \sigma_i$

By Lemma 7, if *row* $\tau' = \rho$ then *row* $U'(\tau_1) = \rho$. So if $U(\rho)(i) = \sigma_i$ then $(U \circ U')(\rho)(i) = \sigma_i$. I.e. if $i \in$ *dom* $U(\tau')$ then $i \in$ *dom* $(U \circ U')(\tau_1)$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i) = \sigma_i$. A similar argument holds if *row* $\tau' = \gamma$.

Therefore, we have shown that $\tau' \sqsubseteq U'(\tau_1)$. Moreover, since $distribute_p$ is commutative (up to variable renaming), we also have $\tau' \sqsubseteq U'(\tau_2)$, where $U', \tau' = distribute_p\ \tau_1\ \tau_2$.

7.4.2 Ordering of merge

Lemma 11: $\tau' \sqsubseteq U'(\tau_1)$, where $U', \tau' = merge_p\ \tau_1\ \tau_2$

In order to show that $\tau' \sqsubseteq U'(\tau_1)$, we must show that for all ground substitutions

U, $dom\ U(\tau') \supseteq dom\ (U \circ U')(\tau_1)$ and $U(\tau')(i) \leq (U \circ U')(\tau_1)(i)$, for all $i \in dom\ (U \circ U')(\tau_1)$, where $U', \tau' = merge_p\ \tau_1\ \tau_2$.

If $i \in dom\ U(\tau')$ then the i-field could have arisen in the same three ways as for $distribute_p$ in Lemma 10.

Proof: Consider each of these three cases in turn:

Case 1: $\tau'(i) = \sigma_i$

By Lemma 3(1), since $\tau'(i) = \sigma_i$, we have $U'(\tau_1)(i) = \sigma_i$ or $U'(\tau_1)(i) =$ **absent**. I.e. if $i \in dom\ (U \circ U')(\tau_1)$ then $i \in dom\ U(\tau')$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i)$.

Case 2: $\tau'(i) = \Delta_i$ and $U(\Delta_i) = \sigma_i$

By Lemma 3(3), since $\tau'(i) = \Delta_i$, we have $U'(\tau_1)(i) = \Delta_i$ or $U'(\tau_1)(i) =$ **absent**. Therefore, if $U(\tau')(i) = \sigma_i$ then $(U \circ U')(\tau_1)(i) = \sigma_i$ or $(U \circ U')(\tau_1)(i) =$ **absent**. I.e. if $i \in dom\ (U \circ U')(\tau_1)$ then $i \in dom\ U(\tau')$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i) = \sigma_i$.

Case 3: $row\ \tau' = \rho$ and $U(\rho)(i) = \sigma_i$

By Lemma 8, if $row\ \tau' = \rho$ then $row\ U'(\tau_1) = \rho$ or $row\ U'(\tau_1) =$ **exactly**. So if $U(\rho)(i) = \sigma_i$ then either $(U \circ U')(\tau_1)(i) = \sigma_i$ or $(U \circ U')(\tau_1)(i) =$ **absent**. I.e. if $i \in dom\ (U \circ U')(\tau_1)$ then $i \in dom\ U(\tau')$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i) = \sigma_i$.

Therefore, we have shown that $\tau' \sqsubseteq U'(\tau_1)$. Moreover, since $merge_p$ is commutative (up to variable renaming), we also have $\tau' \sqsubseteq U'(\tau_2)$, where $U', \tau' = merge_p\ \tau_1\ \tau_2$.

7.4.3 Ordering of select

Lemma 12: $U'(\tau_1) \sqsubseteq \tau'$, where $U', \tau' = select_p\ \tau_1\ \tau_2$

In order to show that $U'(\tau_1) \sqsubseteq \tau'$, we must show that for all ground substitutions U, $dom\ (U \circ U')(\tau_1) \supseteq dom\ U(\tau')$ and $(U \circ U')(\tau_1)(i) \leq U(\tau')(i)$, for all $i \in dom\ U(\tau')$, where $U', \tau' = select_p\ \tau_1\ \tau_2$.

If $i \in dom\ U(\tau')$ then the i-field could have arisen in the same three ways as for $distribute_p$ in Lemma 10.

Proof: Consider each of these three cases in turn:

Case 1: $\tau'(i) = \sigma_i$

By Lemma 5(1), since $\tau'(i) = \sigma_i$, we have $U'(\tau_1)(i) \leq \sigma_i$, i.e. $U'(\tau_1)$ contains a field "i: σ_i'" where $\sigma_i' \leq \sigma_i$. I.e. if $i \in dom\ U(\tau')$ then $i \in dom\ (U \circ U')(\tau_1)$ and $(U \circ U')(\tau_1)(i) \leq U(\tau')(i)$.

Case 2: $\tau'(i) = \Delta_i$ and $U(\Delta_i) = \sigma_i$

By Lemma 5(3), since $\tau'(i) = \Delta_i$, we have $U'(\tau_1)(i) = \Delta_i$. Therefore, if $U(\tau')(i) = \sigma_i$ then $(U \circ U')(\tau_1)(i) = \sigma_i$. I.e. if $i \in dom\ U(\tau')$ then $i \in dom\ (U \circ U')(\tau_1)$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i)$.

Case 3: $row\ \tau' = \rho$ and $U(\rho)(i) = \sigma_i$

By Lemma 7, if $row\ \tau' = \rho$ then $row\ U'(\tau_1) = \rho$. So if $U(\rho)(i) = \sigma_i$ then $(U \circ U')(\rho)(i) = \sigma_i$. I.e. if $i \in dom\ U(\tau')$ then $i \in dom\ (U \circ U')(\tau_1)$ and $U(\tau')(i) = (U \circ U')(\tau_1)(i)$.

Therefore, we have shown that $U'(\tau_1) \sqsubseteq \tau'$. Moreover, since $select_p$ is commutative (up to variable renaming), we also have $U'(\tau_2) \sqsubseteq \tau'$, where $U', \tau' = select_p\ \tau_1\ \tau_2$.

7.5 Soundness Lemmas

The following lemmas about the auxiliary operations are needed in the proof of soundness. The lemmas are stated in terms of binding record sort schemes, but also hold for transient record sort schemes. We omit the proofs of the lemmas due to time constraints. We do, however, provide some intuition for each of them.

Lemma 13: If bindings b is an instance of *distribute* β_1 β_2 under some substitution U, then b is an instance of both β_1 and β_2 under the same substitution U, i.e.

$\forall b, \beta_1, \beta_2, U$ if $b \in U(\beta')$ then $b \in (U \circ U')(\beta_1)$ and $b \in (U \circ U')(\beta_2)$, where $U', \beta' = \mathit{distribute}_p\ \beta_1\ \beta_2$

Intuition: If "i: S_i" is a field in *distribute* β_1 β_2, then S_i is the meet of the sorts of the i-fields in each of β_1 and β_2, say S_1 and S_2 respectively. Therefore, if a datum d is an instance of S_i then d is also an instance of S_1 and S_2, i.e. $d \in (S_1\ \&\ S_2)$ implies $d \in S_1$ and $d \in S_2$. In fact, Lemma 13 follows from Lemma 10.

Lemma 14: If bindings b is not an instance of *distribute* β_1 β_2 under any substitution U, then either b is not an instance of β_1 or b is not an instance of β_2 under the same substitution U, i.e.

$\forall b, \beta_1, \beta_2, U$ if $b \notin U(\beta')$ then $b \notin (U \circ U')(\beta_1)$ or $b \notin (U \circ U')(\beta_2)$, where $U', \beta' = \mathit{distribute}_p\ \beta_1\ \beta_2$

Intuition: If "i: S_i" is a field in *distribute* β_1 β_2, then S_i is the meet of the sorts of the i-fields in each of β_1 and β_2, say S_1 and S_2 respectively. Therefore, if a datum d is not an instance of S_i then d is not instance of either S_1 or S_2, i.e. $d \notin (S_1\ \&\ S_2)$ implies $d \notin S_1$ or $d \notin S_2$.

Lemma 15: If bindings b_1 is an instance of β_1 under some substitution U, then b_1 is an instance of *switch* β_1 β_2 for any β_2 under the same substitution U, i.e.

$\forall b_1, \beta_1, \beta_2, U$ if $b_1 \in (U \circ U')(\beta_1)$ then $b_1 \in U(\beta')$, where $U', \beta' = \mathit{switch}_p\ \beta_1\ \beta_2$

Intuition: If "i: S_i" is a field in *switch* β_1 β_2, then S_i is the join of the sorts of the i-fields in each of β_1 and β_2, say S_1 and S_2 respectively. Therefore, if a datum d is an instance of S_i then d is an instance of either S_1 or S_2, i.e. $d \in (S_1 \mid S_2)$ implies $d \in S_1$ or $d \in S_2$.

Lemma 16: If bindings b is not an instance of *switch* β_1 β_2 under any substitution U, then b is not an instance of β_1 and b is not an instance of β_2 under any substitution U, i.e.

$\forall b, \beta_1, \beta_2, U$ if $b \notin U(\beta')$ then $b \notin (U \circ U')(\beta_1)$ and $b \notin (U \circ U')(\beta_2)$, where $U', \beta' = switch_p\ \beta_1\ \beta_2$

Intuition: If "i: S_i" is a field in *switch* β_1 β_2, then S_i is the join of the sorts of the i-fields in each of β_1 and β_2, say S_1 and S_2 respectively. Therefore, if a datum d is not an instance of S_i then d is not instance of both S_1 and S_2, i.e. $d \notin (S_1 \mid S_2)$ implies $d \notin S_1$ and $d \notin S_2$.

Lemma 17: If bindings b_1 is an instance of β_1 and bindings b_2 is an instance of β_2 under some substitution U, and b_1 and b_2 are mergeable, then *merge* b_1 b_2 is an instance of *merge* β_1 β_2 under the same substitution U, i.e.

$\forall b_1, b_2, \beta_1, \beta_2, U$ if $b_1 \in (U \circ U')(\beta_1)$ and $b_2 \in (U \circ U')(\beta_2)$ and *mergeable* $b_1\ b_2$ then *merge* $b_1\ b_2 \in U(\beta')$, where $U', \beta' = merge_p\ \beta_1\ \beta_2$

Intuition: If "i: S_i" is a field in *merge* β_1 β_2, then S_i is identical to the sort of the i-field in either β_1 or β_2. Also, if β_1 contains the i-field, then it must be absent from β_2 (and vice versa).

Lemma 18: If bindings b_1 is an instance of β_1 and bindings b_2 is an instance of β_2 under some substitution U, then *overlay* b_1 b_2 is an instance of *overlay* β_1 β_2 under the same substitution U, i.e.

$\forall b_1, b_2, \beta_1, \beta_2, U$ if $b_1 \in (U \circ U')(\beta_1)$ and $b_2 \in (U \circ U')(\beta_2)$ then *overlay* $b_1\ b_2 \in U(\beta')$, where $U', \beta' = overlay_p\ \beta_1\ \beta_2$

Intuition: If "i: S_i" is a field in *overlay* β_1 β_2, then S_i is identical to the sort of the i-field in either β_1 or β_2. If β_1 contains the i-field, then S_i is the sort of the i-field in β_1, otherwise, S_i is the sort of the i-field in β_2.

Lemma 19: If bindings b_1 is an instance of β_1 under some substitution U, then b_1 is an instance of *select* β_1 β_2 for any β_2 and the same substitution U, i.e.

$$\forall b_1, \beta_1, \beta_2, U \text{ if } b_1 \in (U \circ U')(\beta_1) \text{ then } b_1 \in U(\beta'), \text{ where } U', \beta' = select_p\ \beta_1\ \beta_2$$

Similarly if bindings b_2 is an instance of β_2, then b_2 is an instance of *select* β_1 β_2 for any β_1 by commutativity.

Intuition: If "i: S_i" is a field in *select* β_1 β_2, then S_i is the join of the sorts of the i-fields in each of β_1 and β_2, say S_1 and S_2 respectively. Therefore, if a datum d is an instance of S_1 then d is also an instance of S_i, i.e. $d \in S_1$ implies $d \in (S_1 \mid S_2)$ for any sort S_2. In fact, Lemma 19 follows from Lemma 12.

7.6 Definition of Soundness

Before proceeding, we must decide exactly what we actually mean by soundness in ACTRESS action notation. Soundness is an essential property of any type system. In ACTRESS action notation, it means the sort inference rules are valid, i.e. they assign sensible sorts to the actions. There is, however, an important point to remember about an action sort scheme: it does not guarantee the successful completion of an action, i.e. it says that, given inputs satisfying the input sort schemes, and if the action completes, then the sort of the outputs will be satisfy the output sort schemes. This differs from soundness in most other type inference algorithms, where soundness guarantees that "well-typed expressions do not go wrong" [Mil78], i.e. the assigning of a type proves that the expression can be evaluated to give a value of that type—the calculation cannot fail. This means our soundness property is weaker than that of, say, Hindley-Milner type inference.

The soundness property, *sound*, is applied to an individual sort judgement and asserts that the sort assigned by that judgement satisfies the soundness criteria. The soundness property has two distinct, although related, forms: one for action sort judgements and one for yielder sort judgements.

7.6.1 Soundness of Action Sort Schemes

Consider an action A that is assigned a sort $(\tau, \beta) \hookrightarrow (\tau', \beta')$ in some environment ε, i.e. we have:

$$\varepsilon \vdash A: (\tau, \beta) \hookrightarrow (\tau', \beta')$$

The sort inferences rules will use other sort judgements and some additional constraints to derive the sort of A.

Now consider a performance of A with input transients t, bindings b, and storage s. This performance of A can complete, fail, or diverge. If the performance of A completes, let it produce output transients t', bindings b', and storage s'.

We want to relate the outcome of A to the sort assigned by the inference rules. This gives us the first soundness criterion for actions:

(1) If t and b are instances of τ and β under a (ground) substitution U, and if A completes, then A will produce transients t' and bindings b' that are instances of τ' and β' under the same substitution U.

Thus if the input transients and bindings match their expected record sort schemes, then the output transients and bindings will also match, i.e. the sort accurately predicts the information produced by performing the action (when it completes). However, what happens if the input transients or bindings do not match their expected sort schemes? This gives us the second soundness criterion for actions:

(2) If t and b are not instances of τ and β under any (ground) substitution U, then the performance of A cannot complete. (Note that for most action primitives, A will fail, but in general, the action may also diverge.)

These criteria are formalised in the following definition of the soundness property for actions:

$sound(\varepsilon \vdash A\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ iff

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C)$ then
   $\forall s$. if $(t, b, s) \vdash A \Rightarrow (completed, t', b', s')$
      then $t' \in U(\tau')$ and $b' \in U(\beta')$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta))$ then
   $\forall s$. $(t, b, s) \vdash A \Rightarrow (o', t', b', s')$ and $o' \neq completed$

Here, C is the set of constraints used to derive the sort judgement. This set of constraints may contain uninstantiated (sort) variables, and so any ground substitution U must also satisfy all of these constraints for the sort judgement to be valid. We denote the satisfaction of the constraints by writing $U(C)$. Actions and yielders involving sub-terms require any constraints from those sub-terms to be satisfied. Therefore, we always take the union of the sets of constraints in the sub-terms.

7.6.2 Soundness of Yielder Sort Schemes

Similarly, consider a yielder Y that is assigned a sort $(\tau, \beta) \rightsquigarrow \sigma$ in some environment ε, i.e. we have:

$\varepsilon \vdash Y\colon (\tau, \beta) \rightsquigarrow \sigma$

Now consider an evaluation of Y with input transients t, bindings b, and storage s. This evaluation of Y can either yield a datum or yield nothing.

We want to relate the datum yielded by Y to the sort assigned by the inference rules. This gives us the first soundness criterion for yielders:

(1) If t and b are instances of τ and β under a (ground) substitution U, and if Y yields a datum d, then d is an instance of σ under the same substitution U.

Thus if the input transients and bindings match their expected record sort schemes, then the datum yielded will also match, i.e. the sort accurately predicts the sort of the datum produced by evaluating the yielder (when it does not yield nothing). However, what happens if the input transients or bindings do not match their expected sort schemes? This gives us the second soundness criterion for yielders:

(2) If t and b are not instances of τ or β for any (ground) substitution U, then the evaluation of Y must yield nothing.

These criteria are formalised in the following definition of the soundness property for yielders:

sound($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow \sigma$, C) iff

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C)$ then
 $\forall s$. if $(t, b, s) \vdash Y \Rightarrow d$ then $d \in U(\sigma)$

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$) then
 $\forall s$. $(t, b, s) \vdash Y \Rightarrow$ nothing

7.7 Proof of Soundness

The proof of soundness is constructed inductively over the structure of actions and yielders.

7.7.1 Basic Action Notation

Case: complete

We have to show that:

sound($\varepsilon \vdash$ complete: $(\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{\ \}, \{\ \}), \{\ \}$), i.e.

(1) $\forall t, b, U$. if $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$ and $U(\{\ \})$ then
 $\forall s$. if $(t, b, s) \vdash$ complete $\Rightarrow$ (*completed*, $\{\ \}$, $\{\ \}$, s)
  then $\{\ \} \in U(\{\ \})$ and $\{\ \} \in U(\{\ \})$

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$) then
 $\forall s$. $(t, b, s) \vdash$ complete $\Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$ and $U(\{\ \})$

By rule (COMPLETE-S1), we have $\forall t, b, s$. $(t, b, s) \vdash$ complete $\Rightarrow$ (*completed*, $\{\ \}$, $\{\ \}$, s). Also, we have $\forall U$. $\{\ \} \in U(\{\ \})$.

Criterion (2):

By rule (COMPLETE-S1), we have $\forall t, b, s$. $(t, b, s) \vdash$ complete $\Rightarrow$ (*completed*, $\{\ \}$, $\{\ \}$, s). Therefore, if we can show that $\forall t, b$. $\exists U$ s.t. $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$, the result follows. Choose transients t and bindings b, now take $U = [\gamma_1 \mapsto t, \gamma_2 \mapsto b]$. Clearly, since $t \in t$ and $b \in b$, we have $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$.

Case: A_1 and then A_2

We have to show that:

sound($\varepsilon \vdash A_1$: $(\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1$) and *sound*($\varepsilon \vdash A_2$: $(\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2$) implies *sound*($\varepsilon \vdash A_1$ and then A_2: (*distribute* τ_1 τ_2, *distribute* β_1 β_2) $\hookrightarrow$ (*merge* τ_1' τ_2', *merge* β_1' β_2'), $C_1 \cup C_2$), i.e.

(1) $\forall t, b, U$. if $t \in U$(*distribute* τ_1 τ_2) and $b \in U$(*distribute* β_1 β_2) and $U(C_1 \cup C_2)$ then
$\forall s$. if $(t, b, s) \vdash A_1$ and then $A_2 \Rightarrow$ (*completed*, *merge* t_1 t_2, *merge* b_1 b_2, s_2) then *merge* t_1 $t_2 \in U$(*merge* τ_1' τ_2') and *merge* b_1 $b_2 \in U$(*merge* β_1' β_2')

(2) $\forall t, b$. if ($\neg\exists$ U s.t. $t \in U$(*distribute* τ_1 τ_2) and $b \in U$(*distribute* β_1 β_2)) then
$\forall s$. $(t, b, s) \vdash A_1$ and then $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $t \in U$(*distribute* τ_1 τ_2) and $b \in U$(*distribute* β_1 β_2) and $U(C_1 \cup C_2)$

By Lemma 13, we have $t \in U(\tau_1)$ and $t \in U(\tau_2)$. Similarly by Lemma 13, we also have $b \in U(\beta_1)$ and $b \in U(\beta_2)$. By *sound*($\varepsilon \vdash A_1$: $(\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1$), we have:

$\forall$ s . if $(t, b, s) \vdash A_1 \Rightarrow$ (*completed*, t_1, b_1, s_1) then $t_1 \in U(\tau_1')$ and $b_1 \in U(\beta_1')$

Similarly, by *sound*($\varepsilon \vdash A_2$: $(\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2$), we have:

$\forall$ s . if $(t, b, s) \vdash A_2 \Rightarrow$ (*completed*, t_2, b_2, s_2) then $t_2 \in U(\tau_2')$ and $b_2 \in U(\beta_2')$

From the semantic rules, we have $(t, b, s) \vdash A_1$ and then $A_2 \Rightarrow$ (*completed*, *merge* $t_1\ t_2$, *merge* $b_1\ b_2$, s_2) implies $(t, b, s) \vdash A_1 \Rightarrow$ (*completed*, t_1, b_1, s_1) and $(t, b, s_1) \vdash A_2 \Rightarrow$ (*completed*, t_2, b_2, s_2), i.e. "A_1 and then A_2" only completes if both A_1 and A_2 complete. Therefore, we have $t_1 \in U(\tau_1')$ and $t_2 \in U(\tau_2')$, and by Lemma 17, we have *merge* $t_1\ t_2 \in U($*merge* $\tau_1'\ \tau_2')$. Similarly by Lemma 17, we also have *merge* $b_1\ b_2 \in U($*merge* $\beta_1'\ \beta_2')$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U($*distribute* $\tau_1\ \tau_2)$ and $b \in U($*distribute* $\beta_1\ \beta_2)$

Assume $t \notin U($*distribute* $\tau_1\ \tau_2)$. By Lemma 14, we have $t \notin U(\tau_1)$ and $t \notin U(\tau_2)$. Therefore, we have either $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*, or $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*. By the semantic rules, we have $(t, b, s) \vdash A_1$ and then $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*. Similarly, if $b \notin U($*distribute* $\beta_1\ \beta_2)$.

Case: A_1 and A_2

This is the similar to "A_1 and then A_2".

Case: A_1 or A_2

We have to show that:

sound$(\varepsilon \vdash A_1: (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1)$ and *sound*$(\varepsilon \vdash A_2: (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2)$ implies *sound*$(\varepsilon \vdash A_1$ or A_2: (*switch* $\tau_1\ \tau_2$, *switch* $\beta_1\ \beta_2$) $\hookrightarrow$ (*select* $\tau_1'\ \tau_2'$, *select* $\beta_1'\ \beta_2'$), $C_1 \cup C_2)$, i.e.

(1) $\forall t, b, U$. if $t \in U($*switch* $\tau_1\ \tau_2)$ and $b \in U($*switch* $\beta_1\ \beta_2)$ and $U(C_1 \cup C_2)$ then
  $\forall s$. if $(t, b, s) \vdash A_1$ or $A_2 \Rightarrow$ (*completed*, t', b', s')
    then $t' \in U($*select* $\tau_1'\ \tau_2')$ and $b' \in U($*select* $\beta_1'\ \beta_2')$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U($*switch* $\tau_1\ \tau_2)$ and $b \in U($*switch* $\beta_1\ \beta_2))$ then
  $\forall s$. $(t, b, s) \vdash A_1$ or $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $U(C_1 \cup C_2)$

From the semantic rules, we have $(t, b, s) \vdash A_1$ or $A_2 \Rightarrow$ (*completed*, t', b', s') implies $(t, b, s) \vdash A_1 \Rightarrow$ (*completed*, t', b', s') or $(t, b, s) \vdash A_2 \Rightarrow$ (*completed*, t', b', s'), i.e. "A_1 or A_2" completes only if either A_1 or A_2 completes.

Assume $(t, b, s) \vdash A_1 \Rightarrow (\textit{completed}, t', b', s')$. From soundness, we have $t \in U(\tau_1)$ and $b \in U(\beta_1)$ (otherwise criterion 2 would have meant A_1 would not complete). Also, by soundness, we have $t' \in U(\tau_1')$ and $b' \in U(\beta_1')$. From Lemma 15, we have $t \in U(\textit{switch}\ \tau_1\ \tau_2)$ and $b \in U(\textit{switch}\ \beta_1\ \beta_2)$, and from Lemma 19, we have $t' \in U(\textit{select}\ \tau_1'\ \tau_2')$ and $b' \in U(\textit{select}\ \beta_1'\ \beta_2')$. Therefore, if A_1 completes, we have that criterion 1 for "A_1 or A_2" holds.

A similar argument applies if A_2 completes.

Criterion (2): $\neg\exists\ U$ s.t. $t \in U(\textit{switch}\ \tau_1\ \tau_2)$ and $b \in U(\textit{switch}\ \beta_1\ \beta_2)$

Assume $t \notin U(\textit{switch}\ \tau_1\ \tau_2)$. By Lemma 16, we have $t \notin U(\tau_1)$ and $t \notin U(\tau_2)$. Therefore, by soundness, we have $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$, and $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$. Therefore, from the semantic rules, we have $(t, b, s) \vdash A_1$ or $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$. Similarly, if $b \notin U(\textit{switch}\ \beta_1\ \beta_2)$.

Case: A_1 else A_2

Since "A_1 else A_2" is an abbreviation for "(check (it is true) then A_1) or (check (it is false) then A_2)", soundness follows immediately.

Case: unfolding A

We have to show that:

$\textit{sound}(\varepsilon \vdash A\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ implies $\textit{sound}(\varepsilon \vdash$ unfolding $A\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ using the sort inference rules (UNFOLDING-I) and (UNFOLD-I).

The proof $\textit{sound}(\varepsilon \vdash$ unfolding $A\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ is more complex than the proofs for the other action notation combinators. Since "unfolding A" represents a family of action terms, whose size depends on the level of the unfolding, we must prove soundness for all such possible levels of unfolding. Using standard techniques[Gue81], we represent "unfolding A" as the union of all possible unfoldings A_∞:

$$\text{unfolding}\ A \approx A_\infty \qquad \text{where}\ A_\infty = \bigsqcup_{i \geq 0} A_i \qquad \begin{cases} A_0 \triangleq \text{diverge} \\ A_{i+1} \triangleq A[A_i\ /\ \text{unfold}] \end{cases}$$

Where "$A_1 \approx A_2$" means A_1 *is operationally equivalent to* A_2, and "$A_1 \triangleq A_2$" means A_1 *is defined as* A_2. For ACTRESS action notation, we can take "A_1 is operationally equivalent to A_2" to mean $\forall t, b, s\ .\ (t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ iff $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$. It follows that if we can prove $sound(\varepsilon \vdash A_\infty\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ then $sound(\varepsilon \vdash \text{unfolding}\ A\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ is immediate.

First, we prove $\forall i \geq 0\ .\ sound(\varepsilon \vdash A_i\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ using mathematical induction.

Case $k = 0$:

We must show $sound(\varepsilon \vdash A_0\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$. Since $A_0 \triangleq$ diverge, and $sound(\varepsilon \vdash \text{diverge}\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ is trivial (this action never completes and so the two criteria are immediately satisified), then we have $sound(\varepsilon \vdash A_0\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$.

Case $k = i + 1$:

We must show $sound(\varepsilon \vdash A_i\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ implies $sound(\varepsilon \vdash A_{i+1}\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$. From the definition, we have $A_{i+1} \triangleq A[A_i\ /\ \text{unfold}]$, and from the inductive hypothesis, we have $sound(\varepsilon \vdash A_i\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$. Now, from rule (UNFOLD-I) we have $[\text{unfold}\colon (\tau, \beta) \hookrightarrow (\tau'\ \beta')]\ \varepsilon \vdash \text{unfold}\colon (\tau, \beta) \hookrightarrow (\tau', \beta')$, and therefore, we are replacing "unfold" with an action of the same sort. Therefore, we are not altering the proof that $\varepsilon \vdash A\colon (\tau, \beta) \hookrightarrow (\tau', \beta')$. So, it follows that $sound(\varepsilon \vdash A\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ and $sound(\varepsilon \vdash A_i\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$ implies $sound(\varepsilon \vdash A[A_i\ /\ \text{unfold}]\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$, and therefore, $sound(\varepsilon \vdash A_{i+1}\colon (\tau, \beta) \hookrightarrow (\tau', \beta'), C)$.

Therefore, by mathematical induction, we have $\forall i \geq 0$, *sound*($\varepsilon \vdash A_i$: $(\tau, \beta) \hookrightarrow (\tau', \beta')$, C).

Now, each of the A_i actions is a subtree of A_∞. Further, the A_i-trees (including their sorts) are consistent: $A_i \leq A_{i+1}$, $\forall i \geq 0$. That is, $\forall i \geq 0$, $(t, b, s) \vdash A_i \Rightarrow$ (*completed*, t', b', s') implies $(t, b, s) \vdash A_{i+1} \Rightarrow$ (*completed*, t', b', s'). So, we define the sort-checked version of A_∞ as $\bigsqcup_{i \geq 0} A_i$ (as A_∞ is the *infinite overlay* of the A_i's).

For all A_i, we have $\varepsilon \vdash A_i$: $(\tau, \beta) \hookrightarrow (\tau', \beta')$, and *sound*($\varepsilon \vdash A_i$: $(\tau, \beta) \hookrightarrow (\tau', \beta')$, C). By definition, we have $\varepsilon \vdash A_\infty$: $(\tau, \beta) \hookrightarrow (\tau', \beta')$, and since the behaviour of A_∞ is equivalent to the union of the behaviours of the A_i's, we have *sound*($\varepsilon \vdash A_\infty$: $(\tau, \beta) \hookrightarrow (\tau', \beta')$, C).

Case: Y_1 is Y_2

We have to show that:

sound($\varepsilon \vdash Y_1$: $(\tau_1, \beta_1) \rightsquigarrow \sigma$, C_1) and *sound*($\varepsilon \vdash Y_2$: $(\tau_2, \beta_2) \rightsquigarrow \sigma$, C_2) implies
sound($\varepsilon \vdash Y_1$ is Y_2: (*distribute* τ_1 τ_2, *distribute* β_1 β_2) $\rightsquigarrow$ truth-value, $C_1 \cup C_2$), i.e.

(1) $\forall t, b, U$. if $t \in U$(*distribute* τ_1 τ_2) and $b \in U$(*distribute* β_1 β_2)
and $U(C_1 \cup C_2)$ then
$\forall s$. if $(t, b, s) \vdash Y_1$ is $Y_2 \Rightarrow d$ then $d \in U$(truth-value)

(2) $\forall t, b$. if ($\neg\exists$ U s.t. $t \in U$(*distribute* τ_1 τ_2) and $b \in U$(*distribute* β_1 β_2)) then
$\forall s$. $(t, b, s) \vdash Y_1$ is $Y_2 \Rightarrow$ nothing

Criterion (1): Assume $t \in U$(*distribute* τ_1 τ_2) and $b \in U$(*distribute* β_1 β_2) and $U(C_1 \cup C_2)$

By Lemma 13, we have $t \in U(\tau_1)$ and $t \in U(\tau_2)$. Similarly by Lemma 13, we also have $b \in U(\beta_1)$ and $b \in U(\beta_2)$. By *sound*($\varepsilon \vdash Y_1$: $(\tau_1, \beta_1) \rightsquigarrow \sigma$, C_1), we have:

$\forall$ s . if $(t, b, s) \vdash Y_1 \Rightarrow d_1$ then $d_1 \in U(\sigma)$

Similarly, by *sound*($\varepsilon \vdash Y_2$: $(\tau_2, \beta_2) \rightsquigarrow \sigma$, C_2), we have:

$\forall s$. if $(t, b, s) \vdash Y_2 \Rightarrow d_2$ then $d_2 \in U(\sigma)$

From rule (IS-S1), we have $d_1 = d_2$ implies d = true, and from rule (IS-S2), we have $d_1 \neq d_2$ implies d = false. Therefore, in both cases, we have $d \in U$(truth-value).

Criterion (2): $\neg\exists\, U$ s.t. $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$

Assume $t \notin U(\textit{distribute}\ \tau_1\ \tau_2)$. By Lemma 14, we have $t \notin U(\tau_1)$ or $t \notin U(\tau_2)$. Therefore, we have either $(t, b, s) \vdash Y_1 \Rightarrow$ nothing or $(t, b, s) \vdash Y_1 \Rightarrow$ nothing. Therefore, we have $(t, b, s) \vdash Y_1$ is $Y_2 \Rightarrow$ nothing. Similarly, if $b \notin U(\textit{distribute}\ \beta_1\ \beta_2)$.

7.7.2 Functional Action Notation

Case: give Y label # n

We have to show that:

sound$(\varepsilon \vdash Y\!: (\tau, \beta) \rightsquigarrow \sigma, \mathcal{C})$ implies *sound*$(\varepsilon \vdash$ give Y label # n: $(\tau, \beta) \hookrightarrow (\{n\!: \sigma\}, \{\ \}), \mathcal{C})$, i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(\mathcal{C})$ then

$\forall s$. if $(t, b, s) \vdash$ give Y label # $n \Rightarrow (\textit{completed}, \{n \mapsto d\}, \{\ \}, s)$

then $\{n \mapsto d\} \in U(\{n\!: \sigma\})$ and $\{\ \} \in U(\{\ \})$

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$) then

$\forall s$. $(t, b, s) \vdash$ give Y label # $n \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(\beta)$ and $U(\mathcal{C})$

By *sound*$(\varepsilon \vdash Y\!: (\tau, \beta) \rightsquigarrow \sigma, \mathcal{C})$, we have $d \in U(\sigma)$. Therefore, from the semantic rules, we have $\forall s$. $(t, b, s) \vdash$ give Y label # $n \Rightarrow (\textit{completed}, \{n \mapsto d\}, \{\ \}, s)$ and $\{n \mapsto d\} \in U(\{n\!: \sigma\})$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$

By *sound*$(\varepsilon \vdash Y\!: (\tau, \beta) \rightsquigarrow \sigma, \mathcal{C})$, we have $(t, b, s) \vdash Y \Rightarrow$ nothing. So, from the semantic rules , we have $(t, b, s) \vdash$ give Y label # $n \Rightarrow (\textit{failed}, \{\ \}, \{\ \}, s)$ and *failed* $\neq$

completed.

Case: check Y

We have to show that:

sound($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow \sigma, C$) implies *sound*($\varepsilon \vdash$ check Y: $(\tau, \beta) \hookrightarrow (\{\ \}, \{\ \})$, $C \cup \{\sigma$ & truth-value $\neq$ nothing$\}$), i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{\sigma$ & truth-value $\neq$ nothing$\})$ then
   $\forall s$. if $(t, b, s) \vdash$ check $Y \Rightarrow$ (*completed*, $\{\ \}, \{\ \}, s$)
      then $\{\ \} \in U(\{\ \})$ and $\{\ \} \in U(\{\ \})$

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$) then
   $\forall s$. $(t, b, s) \vdash$ check $Y \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{\sigma$ & truth-value $\neq$ nothing$\})$

Since, $\{\ \} \in U(\{\ \})$ for all substitutions U and by the semantic rules, we have $\forall s$. if $(t, b, s) \vdash$ check $Y \Rightarrow$ (*completed*, $\{\ \}, \{\ \}, s$) then $\{\ \} \in U(\{\ \})$ and $\{\ \} \in U(\{\ \})$.

Criterion (2): $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$

By *sound*($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow \sigma, C$), we have $(t, b, s) \vdash Y \Rightarrow$ nothing. So, from the semantic rules, we have $(t, b, s) \vdash$ check $Y \Rightarrow$ (*failed*, $\{\ \}, \{\ \}, s$) and *failed* $\neq$ *completed.*

Case: A_1 then A_2

We have to show that:

sound($\varepsilon \vdash A_1$: $(\tau_1, \beta_1) \hookrightarrow (\tau, \beta_1'), C_1$) and *sound*($\varepsilon \vdash A_2$: $(\tau, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2$) implies *sound*($\varepsilon \vdash A_1$ then A_2: (τ_1, *distribute* $\beta_1\ \beta_2$) $\hookrightarrow$ (τ_2', *merge* $\beta_1'\ \beta_2'$), $C_1 \cup C_2$), i.e.

(1) $\forall t, b, U$. if $t \in U(\tau_1)$ and $b \in U$(*distribute* $\beta_1\ \beta_2$) and $U(C_1 \cup C_2)$ then
   $\forall s$. if $(t, b, s) \vdash A_1$ then $A_2 \Rightarrow$ (*completed*, t_2, *merge* $b_1\ b_2$, s_2)
      then $t_2 \in U(\tau_2')$ and *merge* $b_1\ b_2 \in U$(*merge* $\beta_1'\ \beta_2'$)

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\tau_1)$ and $b \in U$(*distribute* $\beta_1\ \beta_2$)) then

$\forall s \,.\, (t, b, s) \vdash A_1 \text{ then } A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$

Criterion (1): Assume $t \in U(\tau_1)$ and $b \in U(\mathit{distribute}\ \beta_1\ \beta_2)$ and $U(C_1 \cup C_2)$

By Lemma 13, we have $b \in U(\beta_1)$ and $b \in U(\beta_2)$. By *sound*$(\varepsilon \vdash A_1: (\tau_1, \beta_1) \hookrightarrow (\tau, \beta_1'), C_1)$, we have:

$\forall s \,.$ if $(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)$ then $t_1 \in U(\tau)$ and $b_1 \in U(\beta_1')$

From the semantic rules, we have $(t, b, s) \vdash A_1 \text{ then } A_2 \Rightarrow (\mathit{completed}, t_2, \mathit{merge}\ b_1\ b_2, s_2)$ implies $(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)$ and $(t_1, b, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)$, i.e. "$A_1$ then A_2" only completes if both A_1 and A_2 complete. Therefore, we have $t_1 \in U(\tau)$ and by *sound*$(\varepsilon \vdash A_2: (\tau, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2)$, we have:

$\forall s \,.$ if $(t_1, b, s) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)$ then $t_2 \in U(\tau_2')$ and $b_2 \in U(\beta_2')$

Therefore, we have $t_2 \in U(\tau_2')$, and by Lemma 17, we have $\mathit{merge}\ b_1\ b_2 \in U(\mathit{merge}\ \beta_1'\ \beta_2')$.

Criterion (2): Assume $\neg\exists\ U$ s.t. $t \in U(\tau_1)$ and $b \in U(\mathit{distribute}\ \beta_1\ \beta_2)$

Assume $t \notin U(\tau_1)$. Therefore by soundness, we have $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$, and by the semantic rules, we have $(t, b, s) \vdash A_1 \text{ then } A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$.

Now assume $b \notin U(\mathit{distribute}\ \beta_1\ \beta_2)$. By Lemma 14, we have $b \notin U(\beta_1)$ and $b \notin U(\beta_2)$. Therefore, we have either $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$, or $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$. By the semantic rules, we have $(t, b, s) \vdash A_1 \text{ then } A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$.

Case: the S # n

We have to show that:

sound$(\varepsilon \vdash \text{the } S \# n\text{: } (\{n\text{: } \theta\}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \theta\ \&\ \sigma, \{\theta\ \&\ \sigma \neq \text{nothing}\})$, i.e.

(1) $\forall t, b, U$. if $t \in U(\{n\colon \theta\}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$ and $U(\{\theta \,\&\, \sigma \neq \text{nothing}\})$ then

$\forall s$. if $(t, b, s) \vdash$ the S # $n \Rightarrow t(n)$ then $t(n) \in U(\theta \,\&\, \sigma)$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U(\{n\colon \theta\}\gamma_1)$ and $b \in U(\{\ \}\gamma_2))$ then

$\forall s$. $(t, b, s) \vdash$ the S # $n \Rightarrow$ nothing

Criterion (1): Assume $t \in U(\{n\colon \theta\}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$ and $U(\{\theta \,\&\, \sigma \neq \text{nothing}\})$

By rule (THE-S1), $t(n) \in S = U(\sigma)$, and $t \in U(\{n\colon \theta\}\gamma_1)$ implies $t(n) \in U(\theta)$. Therefore, $t(n) \in (U(\theta) \,\&\, U(\sigma)) = U(\theta \,\&\, \sigma) \neq$ nothing.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\{n\colon \theta\}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$

$\neg\exists\, U$ s.t. $t \notin U(\{n\colon \theta\}\gamma_1)$ implies $n \notin$ *dom* t, and so $(t, b, s) \vdash$ the S # $n \Rightarrow$ nothing.

Case: it

Follows from *sound*$(\varepsilon \vdash$ the datum # n: $(\{n\colon \theta\}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \theta \,\&$ datum, $\{\theta \,\&$ datum $\neq$ nothing$\})$

7.7.3 Declarative Action Notation

Case: bind k to Y

We have to show that:

sound$(\varepsilon \vdash Y\colon (\tau, \beta) \rightsquigarrow \sigma, C)$ implies *sound*$(\varepsilon \vdash$ bind k to $Y\colon (\tau, \beta) \hookrightarrow (\{\ \}, \{k\colon \sigma\}),$ $C \cup \{$bindable $\&\ \sigma \neq$ nothing$\})$, i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{$bindable $\&\ \sigma \neq$ nothing$\})$ then

$\forall s$. if $(t, b, s) \vdash$ bind k to $Y \Rightarrow$ (*completed*, $\{\ \}$, $\{k \mapsto d\}$, s)

then $\{\ \} \in U(\{\ \})$ and $\{k \mapsto d\} \in U(\{k\colon \sigma\})$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta))$ then

$\forall s$. $(t, b, s) \vdash$ bind k to $Y \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C + \{$bindable $\&\ \sigma \neq$ nothing$\})$

By *sound*$(\varepsilon \vdash Y\colon (\tau, \beta) \rightsquigarrow \sigma, C)$, we have $d \in U(\sigma)$. Therefore, from the semantic rules, we have $\forall s$. if $(t, b, s) \vdash$ bind k to $Y \Rightarrow$ (*completed*, $\{\ \}$, $\{k \mapsto d\}$, s), and

$\{k \mapsto d\} \in U(\{k\colon \sigma\})$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$

By *sound*($\varepsilon \vdash Y\colon (\tau, \beta) \rightsquigarrow \sigma, C$), we have $(t, b, s) \vdash Y \Rightarrow$ nothing. So, we have $(t, b, s) \vdash$ bind k to $Y \Rightarrow$ (*failed*, { }, { }, s) and *failed* $\neq$ *completed.*

Case: recursively bind *k* to *Y*

We have to show that:

sound($\varepsilon \vdash Y\colon (\tau,$ *overlay* $\{k\colon \sigma\}\ \beta) \rightsquigarrow \sigma, C$) implies *sound*($\varepsilon \vdash$ recursively bind k to $Y\colon (\tau, \beta) \hookrightarrow (\{\ \}, \{k\colon \sigma\}), C \cup \{$bindable & $\sigma \neq$ nothing$\}$), i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{$bindable & $\sigma \neq$ nothing$\})$ then
  $\forall s$. if $(t, b, s) \vdash$ recursively bind k to $Y \Rightarrow$ (*completed*, { }, $\{k \mapsto d\}, s$)
    then $\{\ \} \in U(\{\ \})$ and $\{k \mapsto d\} \in U(\{k\colon \sigma\})$

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$) then
  $\forall s$. $(t, b, s) \vdash$ recursively bind k to $Y \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{$bindable & $\sigma \neq$ nothing$\})$

From Lemma 18, we have $b \in U(\beta)$ and $\{k \mapsto d\} \in U(\{k\colon \sigma\})$ implies *overlay* $\{k \mapsto d\}\ b \in U($*overlay* $\{k\colon \sigma\}\ \beta)$. By *sound*($\varepsilon \vdash Y\colon (\tau,$ *overlay* $\{k\colon \sigma\}\ \beta) \rightsquigarrow \sigma, C$), we have $(t,$ *overlay* $\{k \mapsto d\}\ b, s) \vdash Y \Rightarrow d$, and $d \in U(\sigma)$. So, from the semantic rules, we have $(t, b, s) \vdash$ recursively bind k to $Y \Rightarrow$ (*completed*, { }, $\{k \mapsto d\}, s$) and $\{\ \} \in U(\{\ \})$ and $\{k \mapsto d\} \in U(\{k\colon \sigma\})$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$

Assume $t \notin U(\tau)$. By soundness, we have $(t,$ *overlay* $\{k \mapsto d\}\ b, s) \vdash Y \Rightarrow$ nothing, and by the semantic rules, we have $(t, b, s) \vdash$ recursively bind k to $Y \Rightarrow$ (*failed*, { }, { }, s) and *failed* $\neq$ *completed.*

Assume $b \notin U(\beta)$, this implies there is some field k' ($\neq k$) for which $b(k') \notin U(\beta)(k')$. Therefore, we have *overlay* $\{k \mapsto d\}\ b \notin U($*overlay* $\{k\colon \sigma\}\ \beta)$. Therefore, by soundness, we have $(t, \mathit{overlay}\ \{k \mapsto d\}\ b, s) \vdash Y \Rightarrow$ nothing, and by the semantic rules, we have $(t, b, s) \vdash$ recursively bind k to $Y \Rightarrow$ $(\mathit{failed}, \{\ \}, \{\ \}, s)$ and $\mathit{failed} \neq$ *completed.*

Case: A_1 hence A_2

We have to show that:

$\mathit{sound}(\varepsilon \vdash A_1{:}\ (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta), C_1)$ and $\mathit{sound}(\varepsilon \vdash A_2{:}\ (\tau_2, \beta) \hookrightarrow (\tau_2', \beta_2'), C_2)$ implies $\mathit{sound}(\varepsilon \vdash A_1$ hence $A_2{:}\ (\mathit{distribute}\ \tau_1\ \tau_2, \beta_1) \hookrightarrow (\mathit{merge}\ \tau_1'\ \tau_2', \beta_2'), C_1 \cup C_2)$, i.e.

(1) $\forall t, b, U$. if $t \in U(\mathit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\beta_1)$ and $U(C_1 \cup C_2)$ then
 $\forall s$. if $(t, b, s) \vdash A_1$ hence $A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, b_2, s_2)$
  then $\mathit{merge}\ t_1\ t_2 \in U(\mathit{merge}\ \tau_1'\ \tau_2')$ and $b_2 \in U(\beta_2')$

(2) $\forall t, b$. if $(\neg\exists\ U$ s.t. $t \in U(\mathit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\beta_1))$ then
 $\forall s$. $(t, b, s) \vdash A_1$ hence $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \mathit{completed}$

Criterion (1): Assume $t \in U(\mathit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\beta_1)$ and $U(C_1 \cup C_2)$

By Lemma 13, we have $t \in U(\tau_1)$ and $t \in U(\tau_2)$. By $\mathit{sound}(\varepsilon \vdash A_1{:}\ (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta), C_1)$, we have:

$\forall\ s$. if $(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)$ then $t_1 \in U(\tau_1')$ and $b_1 \in U(\beta)$

From the semanitc rules, we have $(t, b, s) \vdash A_1$ hence $A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, b_2, s_2)$ implies $(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)$ and $(t, b_1, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)$, i.e. "$A_1$ hence A_2" only completes if both A_1 and A_2 complete. Therefore $b_1 \in U(\beta)$ and by $\mathit{sound}(\varepsilon \vdash A_2{:}\ (\tau_2, \beta) \hookrightarrow (\tau_2', \beta_2'), C_2)$, we have:

$\forall\ s$. if $(t, b_1, s) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)$
 then $t_2 \in U(\tau_2')$ and $b_2 \in U(\beta_2')$

Therefore, by Lemma 17, we have $\mathit{merge}\ t_1\ t_2 \in U(\mathit{merge}\ \tau_1'\ \tau_2')$, and we have $b_2 \in U(\beta_2')$.

Criterion (2): Assume $\neg\exists$ U s.t. $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\beta_1)$

Assume $t \notin U(\textit{distribute}\ \tau_1\ \tau_2)$. By Lemma 14, we have $t \notin U(\tau_1)$ or $t \notin U(\tau_2)$. Therefore, we have either $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$, or $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$. By the semantic rules, we have $(t, b, s) \vdash A_1$ hence $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$.

Now assume $b \notin U(\beta_1)$. Therefore, we have $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$, and by the semantic rules, we have $(t, b, s) \vdash A_1$ hence $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$.

Case: A_1 moreover A_2

We have to show that:

$\textit{sound}(\varepsilon \vdash A_1\text{: } (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1)$ and $\textit{sound}(\varepsilon \vdash A_2\text{: } (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2)$ implies $\textit{sound}(\varepsilon \vdash A_1$ moreover A_2: $(\textit{distribute}\ \tau_1\ \tau_2, \textit{distribute}\ \beta_1\ \beta_2) \hookrightarrow (\textit{merge}\ \tau_1'\ \tau_2', \textit{overlay}\ \beta_2'\ \beta_1'), C_1 \cup C_2)$, i.e.

(1) $\forall t, b, U$. if $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$
and $U(C_1 \cup C_2)$ then
$\forall s$. if $(t, b, s) \vdash A_1$ moreover $A_2 \Rightarrow$
$(\textit{completed}, \textit{merge}\ t_1\ t_2, \textit{overlay}\ b_2\ b_1, s_2)$
then $\textit{merge}\ t_1\ t_2 \in U(\textit{merge}\ \tau_1'\ \tau_2')$ and
$\textit{overlay}\ b_2\ b_1 \in U(\textit{overlay}\ \beta_2'\ \beta_1')$

(2) $\forall t, b$. if $(\neg\exists$ U s.t. $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2))$ then
$\forall s$. $(t, b, s) \vdash A_1$ moreover $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$

Criterion (1): Assume $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$ and $U(C_1 \cup C_2)$

By Lemma 13, we have $t \in U(\tau_1)$ and $t \in U(\tau_2)$. Similarly by Lemma 13, we also have $b \in U(\beta_1)$ and $b \in U(\beta_2)$. By $\textit{sound}(\varepsilon \vdash A_1\text{: } (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1)$, we have:

$\forall$ s . if $(t, b, s) \vdash A_1 \Rightarrow (\textit{completed}, t_1, b_1, s_1)$ then $t_1 \in U(\tau_1')$ and $b_1 \in U(\beta_1')$

Similarly, by $sound(\varepsilon \vdash A_2\colon (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'), C_2)$, we have:

$$\forall\, s\ .\ \text{if } (t, b, s) \vdash A_2 \Rightarrow (completed, t_2, b_2, s_2) \text{ then } t_2 \in U(\tau_2') \text{ and } b_2 \in U(\beta_2')$$

From the semantic rules, we have $(t, b, s) \vdash A_1$ moreover $A_2 \Rightarrow (completed, merge\ t_1\ t_2, overlay\ b_2\ b_1, s_2)$ implies $(t, b, s) \vdash A_1 \Rightarrow (completed, t_1, b_1, s_1)$ and $(t, b, s_1) \vdash A_2 \Rightarrow (completed, t_2, b_2, s_2)$, i.e. "$A_1$ moreover A_2"completes only if both A_1 and A_2 complete. Therefore, we have $t_1 \in U(\tau_1')$ and $t_2 \in U(\tau_2')$, and by Lemma 17, we have $merge\ t_1\ t_2 \in U(merge\ \tau_1'\ \tau_2')$. By Lemma 18, we also have $overlay\ b_2\ b_1 \in U(overlay\ \beta_2'\ \beta_1')$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(distribute\ \tau_1\ \tau_2)$ and $b \in U(distribute\ \beta_1\ \beta_2)$

Assume $t \notin U(distribute\ \tau_1\ \tau_2)$. By Lemma 14, we have $t \notin U(\tau_1)$ or $t \notin U(\tau_2)$. Therefore, we have either $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq completed$, or $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$ and $o' \neq completed$. By the semantic rules, we have $(t, b, s) \vdash A_1$ moreover $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq completed$. Similarly, if $b \notin U(distribute\ \beta_1\ \beta_2)$.

Case: furthermore A

We have to show that:

$sound(\varepsilon \vdash A\colon (\tau, \beta) \hookrightarrow (\tau', \beta')\ C)$ implies $sound(\varepsilon \vdash$ furthermore $A\colon$
$(\tau, distribute\ \{\ \}\rho\ \beta) \hookrightarrow (\tau', overlay\ \beta'\ \{\ \}\rho), C)$, i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(distribute\ \{\ \}\rho\ \beta)$ and $U(C)$ then
$\forall s$. if $(t, b, s) \vdash$ furthermore $A \Rightarrow (completed, t', overlay\ b'\ b, s')$
then $t' \in U(\tau')$ and $overlay\ b'\ b \in U(overlay\ \beta'\ \{\ \}\rho)$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(distribute\ \{\ \}\rho\ \beta))$ then
$\forall s$. $(t, b, s) \vdash$ furthermore $A \Rightarrow (o', t', b', s')$ and $o' \neq completed$

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(distribute\ \{\ \}\rho\ \beta)$ and $U(C)$

By Lemma 13, we have $b \in U(\{\ \}\rho)$ and $b \in U(\beta)$. By $sound(\varepsilon \vdash A\colon \tau, \beta) \hookrightarrow (\tau', \beta')\ C)$, we have:

$\forall\, s$. if $(t, b, s) \vdash A \Rightarrow$ $(completed, t', b', s')$ then $t' \in U(\tau')$ and $b' \in U(\beta')$

From rule (FURTHERMORE-S1), we have $(t, b, s) \vdash$ furthermore $A \Rightarrow$ $(completed, t', overlay\ b'\ b, s')$ implies $(t, b, s) \vdash A \Rightarrow (completed, t', b', s')$, i.e. "furthermore A" only completes if A completes. Therefore, we have $t' \in U(\tau')$, and by Lemma 18, we have $overlay\ b'\ b \in U(overlay\ \beta'\ \{\ \}\rho)$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(distribute\ \{\ \}\rho\ \beta)$

Assume $t \notin U(\tau)$. Therefore by soundness, we have $(t, b, s) \vdash A \Rightarrow (o', t', b', s')$ and $o' \neq completed$, and by the semantic rules, we have $(t, b, s) \vdash$ furthermore $A \Rightarrow (o', t', b', s')$ and $o' \neq completed.$

Now assume $b \notin U(distribute\ \{\ \}\rho\ \beta)$. Since $distribute\ \{\ \}\rho\ \beta = \beta$, we have $b \notin U(\beta)$. Therefore, we have $(t, b, s) \vdash A \Rightarrow (o', t', b', s')$ and $o' \neq completed.$ By the semantic rules, we have $(t, b, s) \vdash$ furthermore $A \Rightarrow (o', t', b', s')$ and $o' \neq completed.$

Case: A_1 before A_2

We have to show that:

$sound(\varepsilon \vdash A_1$: $(\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1)$ and $sound(\varepsilon \vdash A_2$: $(\tau_2, overlay\ \beta_1'\ \{\ \}\rho) \hookrightarrow (\tau_2', \beta_2'), C_2)$ implies $sound(\varepsilon \vdash A_1$ before A_2: $(distribute\ \tau_1\ \tau_2, distribute\ \{\ \}\rho\ \beta_1) \hookrightarrow (merge\ \tau_1'\ \tau_2', overlay\ \beta_2'\ \beta_1'), C_1 \cup C_2)$, i.e.

(1) $\forall t, b, U$. if $t \in U(distribute\ \tau_1\ \tau_2)$ and $b \in U(distribute\ \{\ \}\rho\ \beta_1)$
and $U(C_1 \cup C_2)$ then
$\forall s$. if $(t, b, s) \vdash A_1$ before $A_2 \Rightarrow (completed, merge\ t_1\ t_2, overlay\ b_2\ b_1, s_2)$
then $merge\ t_1\ t_2 \in U(merge\ \tau_1'\ \tau_2')$ and
$overlay\ b_2\ b_1 \in U(overlay\ \beta_2'\ \beta_1')$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U(distribute\ \tau_1\ \tau_2)$ and $b \in U(distribute\ \{\ \}\rho\ \beta_1))$ then
$\forall s$. $(t, b, s) \vdash A_1$ before $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq completed$

Criterion (1): Assume $t \in U(distribute\ \tau_1\ \tau_2)$ and $b \in U(distribute\ \{\ \}\rho\ \beta_1)$ and $U(C_1 \cup C_2)$

By Lemma 13, we have $t \in U(\tau_1)$ and $t \in U(\tau_2)$. Similarly by Lemma 13, we also have $b \in U(\{\ \}\rho)$ and $b \in U(\beta_1)$. By $sound(\varepsilon \vdash A_1\colon (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'), C_1)$, we have:

$$\forall\, s\ .\ \text{if } (t, b, s) \vdash A_1 \Rightarrow (completed, t_1, b_1, s_1) \text{ then } t_1 \in U(\tau_1') \text{ and } b_1 \in U(\beta_1')$$

Since $b_1 \in U(\beta_1')$ and $b \in U(\{\ \}\rho)$, by Lemma 18 we have *overlay* b_1 $b \in$ *overlay* β_1' $\{\ \}\rho$. Now by $sound(\varepsilon \vdash A_2\colon (\tau_2, overlay\ \beta_1'\ \{\ \}\rho) \hookrightarrow (\tau_2', \beta_2'), C_2)$, we have:

$$\forall\, s\ .\ \text{if } (t, overlay\ b_1\ b, s) \vdash A_2 \Rightarrow (completed, t_2, b_2, s_2)$$
$$\text{then } t_2 \in U(\tau_2') \text{ and } b_2 \in U(\beta_2')$$

From the semantic rules, we have $(t, b, s) \vdash A_1$ before $A_2 \Rightarrow (completed, merge\ t_1\ t_2, overlay\ b_2\ b_1, s_2)$ implies $(t, b, s) \vdash A_1 \Rightarrow (completed, t_1, b_1, s_1)$ and $(t, overlay\ b_1\ b, s_1) \vdash A_2 \Rightarrow (completed, t_2, b_2, s_2)$, i.e. "$A_1$ before A_2"only completes if both A_1 and A_2 complete. Therefore, we have $t_1 \in U(\tau_1')$ and $t_2 \in U(\tau_2')$, and by Lemma 17, we have *merge* $t_1\ t_2 \in U(merge\ \tau_1'\ \tau_2')$. By Lemma 18, we also have *overlay* $b_2\ b_1 \in U(overlay\ \beta_2'\ \beta_1')$.

Criterion (2): Assume $\neg\exists\ U$ s.t. $t \in U(distribute\ \tau_1\ \tau_2)$ and $b \in U(distribute\ \{\ \}\rho\ \beta_1)$

Assume $t \notin U(distribute\ \tau_1\ \tau_2)$. By Lemma 14, we have $t \notin U(\tau_1)$ or $t \notin U(\tau_2)$. Therefore, we have either $(t, b, s) \vdash A_1 \Rightarrow (o', t', b', s')$ and $o' \neq completed$, or $(t, b, s) \vdash A_2 \Rightarrow (o', t', b', s')$ and $o' \neq completed$. By the semantic rules, we have $(t, b, s) \vdash A_1$ before $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq completed$.

Now assume $b \notin U(distribute\ \{\ \}\rho\ \beta_1)$. Since *distribute* $\{\ \}\rho\ \beta_1 = \beta_1$, we have $b \notin U(\beta_1)$. Therefore, we have $(t, b, s) \vdash A \Rightarrow (o', t', b', s')$ and $o' \neq completed$. By the semantic rules, we have $(t, b, s) \vdash A_1$ before $A_2 \Rightarrow (o', t', b', s')$ and $o' \neq completed$.

Case: the S bound to k

We have to show that:

$sound(\varepsilon \vdash \text{the } S \text{ bound to } k\colon (\{\ \}\gamma_1, \{k\colon \theta\}\gamma_2) \rightsquigarrow \theta\ \&\ \sigma, \{\theta\ \&\ \sigma \neq \text{nothing}\})$, i.e.

(1) $\forall t, b, U$. if $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{k\text{: } \theta\}\gamma_2)$ and $U(\{\theta \ \&\ \sigma \neq \text{nothing}\})$ then

$\forall s$. if $(t, b, s) \vdash$ the S bound to $k \Rightarrow b(k)$ then $b(k) \in U(\theta \ \&\ \sigma)$

(2) $\forall t, b$. if $(\neg\exists\ U$ s.t. $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{k\text{: } \theta\}\gamma_2))$ then

$\forall s$. $(t, b, s) \vdash$ the S bound to $k \Rightarrow$ nothing

Criterion (1): Assume $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{k\text{: } \theta\}\gamma_2)$ and $U(\{\theta \ \&\ \sigma \neq \text{nothing}\})$

By rule (BOUND-S1), $b(k) \in S = U(\sigma)$, and $b \in U(\{k\text{: } \theta\}\gamma_2)$ implies $b(k) \in U(\theta)$. Therefore, $b(k) \in (U(\theta) \ \&\ U(\sigma)) = U(\theta \ \&\ \sigma) \neq$ nothing.

Criterion (2): Assume $\neg\exists\ U$ s.t. $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{k\text{: } \theta\}\gamma_2)$

$\neg\exists\ U$ s.t. $b \in U(\{k\text{: } \theta\}\gamma_2)$ implies $k \notin dom\ b$, and so $(t, b, s) \vdash$ the S bound to $k \Rightarrow$ nothing.

7.7.4 Imperative Action Notation

Case: allocate S

We have to show that:

sound($\varepsilon \vdash$ allocate S: $(\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{0\text{: cell}[\sigma]\}, \{\ \}), \{\ \})$, i.e.

(1) $\forall t, b, U$. if $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$ and $U(\{\ \})$ then

$\forall s$. if $(t, b, s) \vdash$ allocate $S \Rightarrow$

$(completed, \{0 \mapsto c\}, \{\ \}, modify\ c$ uninitialized $s)$

then $\{0 \mapsto c\} \in U(\{0\text{: cell}[\sigma]\})$ and $\{\ \} \in U(\{\ \})$

(2) $\forall t, b$. if $(\neg\exists\ U$ s.t. $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2))$ then

$\forall s$. $(t, b, s) \vdash$ allocate $S \Rightarrow (o', t', b', s')$ and $o' \neq completed$

Criterion (1): Assume $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$ and $U(\{\ \})$

By rule (ALLOCATE-S1), we have $c \in S = U(\text{cell}[\sigma])$. Therefore, $\{0 \mapsto c\} \in U(\{0\text{: cell}[\sigma]\})$.

Criterion (2):

By rule (ALLOCATE-S1), we have $\forall t, b, s \,.\, (t, b, s) \vdash$ allocate $S \Rightarrow$ (*completed*, $\{0 \mapsto c\}$, $\{\ \}$, *modify* c uninitialized s). Therefore, if we can show that $\forall t, b \,.\, \exists U$ s.t. $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$, the result follows. Choose transients t and bindings b, now take $U = [\gamma_1 \mapsto t, \gamma_2 \mapsto b]$. Clearly, since $t \in t$ and $b \in b$, we have $t \in U(\{\ \}\gamma_1)$ and $b \in U(\{\ \}\gamma_2)$.

Case: deallocate Y

We have to show that:

sound($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow$ cell[σ], C) implies *sound*($\varepsilon \vdash$ deallocate Y: $(\tau, \beta) \hookrightarrow (\{\ \}, \{\ \})$, C), i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C)$ then
    $\forall s$. if $(t, b, s) \vdash$ deallocate $Y \Rightarrow$ (*completed*, $\{\ \}$, $\{\ \}$, *remove c s*)
        then $\{\ \} \in U(\{\ \})$ and $\{\ \} \in U(\{\ \})$

(2) $\forall t, b$. if $(\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta))$ then
    $\forall s$. $(t, b, s) \vdash$ deallocate $Y \Rightarrow (o', t', b', s')$ and $o' \neq$ *completed*

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C)$

By *sound*($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow$ cell[σ], C) and rule (DEALLOCATE-S1), we have $c \in U(\sigma)$. Also, $\{\ \} \in U(\{\ \})$ for all substitutions U.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$

By *sound*($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow \sigma$, C), we have $(t, b, s) \vdash Y \Rightarrow$ nothing. So, we have $(t, b, s) \vdash$ deallocate $Y \Rightarrow$ (*failed*, $\{\ \}$, $\{\ \}$, s) and *failed* $\neq$ *completed.*

Case: store Y_1 in Y_2

We have to show that:

sound($\varepsilon \vdash Y_1$: $(\tau_1, \beta_1) \rightsquigarrow \sigma_1$, C_1) and *sound*($\varepsilon \vdash Y_2$: $(\tau_2, \beta_2) \rightsquigarrow$ cell[σ_2], C_2) implies *sound*($\varepsilon \vdash$ store Y_1 in Y_2: (*distribute* τ_1 τ_2, *distribute* β_1 β_2) $\hookrightarrow (\{\ \}, \{\ \})$, $C_1 \cup C_2 \cup \{\sigma_1 \,\&\, \sigma_2 \neq \text{nothing}\}$), i.e.

(1) $\forall t, b, U$. if $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$

and $U(C_1 \cup C_2 + \{\sigma_1\ \&\ \sigma_2 \neq \text{nothing}\})$ then

$\forall s$. if $(t, b, s) \vdash$ store Y_1 in $Y_2 \Rightarrow (\textit{completed}, \{\ \}, \{\ \}, \textit{modify}\ c\ d\ s)$

then $\{\ \} \in U(\{\ \})$ and $\{\ \} \in U(\{\ \})$

(2) $\forall t, b$. if ($\neg\exists$ U s.t. $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$) then

$\forall s$. $(t, b, s) \vdash$ store Y_1 in $Y_2 \Rightarrow (o', t', b', s')$ and $o' \neq \textit{completed}$

Criterion (1): Assume $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$ and $U(C_1 \cup C_2 + \{\sigma_1\ \&\ \sigma_2 \neq \text{nothing}\})$

By Lemma 13, we have $t \in U(\tau_1)$ and $t \in U(\tau_2)$. Similarly by Lemma 13, we also have $b \in U(\beta_1)$ and $b \in U(\beta_2)$. By $\textit{sound}(\varepsilon \vdash Y_1\text{: } (\tau_1, \beta_1) \rightsquigarrow \sigma_1, C_1)$, we have:

$\forall s$. if $(t, b, s) \vdash Y_1 \Rightarrow d$ then $d \in U(\sigma_1)$

Similarly, by $\textit{sound}(\varepsilon \vdash Y_2\text{: } (\tau_2, \beta_2) \rightsquigarrow \text{cell}[\sigma_2], C_2)$, we have:

$\forall s$. if $(t, b, s) \vdash Y_2 \Rightarrow c$ then $c \in U(\text{cell}[\sigma_2])$

By rule (STORE-S1), we have $(t, b, s) \vdash$ store Y_1 in $Y_2 \Rightarrow (\textit{completed}, \{\ \}, \{\ \}, \textit{modify}\ c\ d\ s)$ implies $c \in \text{cell}[S]$ and $d \in S$, i.e. "store Y_1 in Y_2" only completes if c is a cell capable of holding a value d.

Criterion (2): Assume $\neg\exists$ U s.t. $t \in U(\textit{distribute}\ \tau_1\ \tau_2)$ and $b \in U(\textit{distribute}\ \beta_1\ \beta_2)$

Assume $\forall U$. $t \notin U(\textit{distribute}\ \tau_1\ \tau_2)$. By Lemma 14, we have $\forall U$. $t \notin U(\tau_1)$ or $\forall U$. $t \notin U(\tau_2)$. Therefore by soundness, we have either $(t, b, s) \vdash Y_1 \Rightarrow$ nothing or $(t, b, s) \vdash Y_2 \Rightarrow$ nothing. Therefore, we have $(t, b, s) \vdash$ store Y_1 in $Y_2 \Rightarrow (\textit{failed}, \{\ \}, \{\ \}, s')$ and $\textit{failed} \neq \textit{completed}$. Similarly if $\forall U$. $b \notin U(\textit{distribute}\ \beta_1\ \beta_2)$.

Case: the S_1 stored in Y_2

We have to show that:

$\textit{sound}(\varepsilon \vdash Y_2\text{: } (\tau, \beta) \rightsquigarrow \text{cell}[\sigma_2], C)$ implies $\textit{sound}(\varepsilon \vdash$ the S stored in $Y\text{: } (\tau, \beta) \rightsquigarrow$

σ_1 & σ_2, $C \cup \{\sigma_1$ & $\sigma_2 \neq$ nothing$\})$, i.e.

(1) $\forall t, b, U$. if $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{\sigma_1$ & $\sigma_2 \neq$ nothing$\})$ then
$\forall s$. if $(t, b, s) \vdash$ the S_1 stored in $Y_2 \Rightarrow s(c)$ then $s(c) \in U(\sigma_1$ & $\sigma_2)$

(2) $\forall t, b$. if ($\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$) then
$\forall s$. $(t, b, s) \vdash$ the S_1 stored in $Y_2 \Rightarrow$ nothing

Criterion (1): Assume $t \in U(\tau)$ and $b \in U(\beta)$ and $U(C \cup \{\sigma_1$ & $\sigma_2 \neq$ nothing$\})$

By *sound*($\varepsilon \vdash Y$: $(\tau, \beta) \rightsquigarrow$ cell$[\sigma_2]$, C) we have $c \in U($cell$[\sigma_2])$, and from rule (STORED-S1) we have $c \in$ *dom s*, which implies $s(c) \in U(\sigma_2)$. Also from rule (STORED-S1) we have $s(c) \in S = U(\sigma_1)$. Therefore, $s(c) \in U(\sigma_2)$ & $U(\sigma_1) = U(\sigma_1$ & $\sigma_2)$.

Criterion (2): Assume $\neg\exists\, U$ s.t. $t \in U(\tau)$ and $b \in U(\beta)$

By *sound*($\varepsilon \vdash Y_2$: $(\tau, \beta) \rightsquigarrow \sigma_2$, C), we have $(t, b, s) \vdash Y_2 \Rightarrow$ nothing. So, we have $(t, b, s) \vdash$ the S_1 stored in $Y_2 \Rightarrow$ nothing.

7.7.5 Reflective Action Notation

Currently, we have chosen to omit the reflective action notation from the proof of soundness. This is because of the current limitations on abstraction sorts discussed in Section 6.3.6, i.e. that abstraction sorts are restricted to being monomorphic. Once the sort inference rules have been re-formulated to eliminate this restriction, we believe the proof of soundness can be extended to include the reflective action notation. We consider it unwise to spend time proving the soundness of the existing sort inference rules when they are about to be changed. We do believe, however, that the existing sort inference rules are also sound, although perhaps with some minor restrictions on actions. For example, we believe it is necessary to forbid abstractions from being storable, as we would have to prove that the sort of an unknown abstraction fetched from the store was sound.

7.8 Conclusion

In this chapter, we have developed a soundness property for ACTRESS action notation, and proved that this property holds for the majority of our sort inference rules. In fact, we believe that this property holds for all of the sort inference rules. We have chosen, however, not to include reflective action notation, as these sort inference rules need to be improved to allow polymorphic abstraction sorts.

The soundness property proves that the input transients and bindings received by an action must match their inferred sorts, if the action is to have a chance of completion. If either the transients or bindings do not match, then the action cannot complete. Furthermore, the soundness property proves that if an action completes, then the transients and bindings that it produces will also match their inferred sorts.

In addition, we have established a number of lemmas regarding the auxiliary operations, and used these to prove the ordering properties of the auxiliary operations proposed in Section 6.2.6.

However, as we discussed in Section 6.2.4, there are limitations on the actions for which we can infer sorts. In particular, there are actions that may complete when performed, but for which we cannot infer a sort. This means, of course, that our sort inference algorithm is not *complete*—we cannot infer a sort for every action that may complete when performed. For example, we cannot infer a sort for the action “(bind "x" to 1) moreover rebind”, since we cannot represent the record sort scheme for the bindings produced by this action. However, this action completes when performed. The sort inference algorithm, therefore, could still be enhanced to infer a sort for every action that may complete.

Chapter 8

Conclusion

8.1 Action Semantics Directed Compiler Generation

The ACTRESS system has been used to generate compilers for a small declarative language, and a small imperative language. Experiments have shown [Mou93a] that the compilation time of a generated compiler is usually within an order of magnitude of a hand-written compiler. Also, the run time of the object code is initially between one and two orders of magnitude slower. However, after applying the action transformations developed by Moura[Mou93a], the run times improve between a factor of 2 and a factor of 10.

These timings compare extremely well with compiler generation systems using other semantic formalisms. Among systems using action semantics, ACTRESS does better than the CANTOR system, but less well than the newer OASIS system. However, the OASIS system has been specifically engineered for the quality of the code generation, where three of its analysis phases are concerned solely with code generation. We believe that if traditional compiler optimisations were added to the code generator in ACTRESS, then ACTRESS too would get within the desired one order of magnitude penalty. However, ACTRESS continues to be the only system that can achieve these timings and still accept actions that require run-time sort checking. This substantially increases the suitability of ACTRESS as the basis of an industrial-strength compiler generation system.

From these three systems, ACTRESS, CANTOR and OASIS, it has been repeatedly demonstrated that an action semantics compiler generation system is potentially suitable for generating usable compilers. No other system based on another formalism has given equivalent results from unmodified, automatically-generated compilers.

8.2 Sort Inference

Our sort inference algorithm represents one of the most complex analyses of action notation. The inferred sort of an action gives precise information about the domains of the transients and bindings required by the action, and the domains of the output transients and bindings produced if the action completes. It is also able to infer individual sorts (i.e. values) in a large number of cases, and propagate these values to the places they are used. This is an important feature that enables the action transformations performed by Moura[Mou93a] to take place.

Our system compares favourably with other systems that perform sort analysis of action notation. The ACTRESS subset of action notation is substantially larger than that used by Even and Schmidt[ES90], and includes important features such as non-deterministic choice ("or"), iteration ("unfolding"), and abstractions. Palsberg[Pal92a,Pal92b] and Ørbæk[Ørb93,Ørb94] use essentially the same subset of action notation as each other. Their subsets, however, avoid the problems of abstraction sorts by restricting the syntax for abstractions to only allow "closure abstraction[D] A" and "enact (A with Y)"[1], where D represents the sort of transient data that the abstraction expects. Moreover, their sort analyses do not allow actions that require run-time sort checks. This means that their systems only accept a specification of a programming language that is both statically-bound and statically-typed. Also, Palsberg's subset restricts "unfolding" actions to be tail-recursive. None of these restrictions are found in our sort inference of ACTRESS action notation.

1. Or rather "enact application A to Y", which is the standard action notation equivalent.

We have shown that our sort inference algorithm is sound with respect to the semantics of action notation for the majority of ACTRESS action notation. We believe that the soundness proof can be extended to include all of ACTRESS action notation. However, possible future work on abstraction sorts will require that part of the soundness proof to be re-formulated, and so we have chosen not to consider the soundness of abstractions at this time.

8.3 Further Work

8.3.1 Improvements

The sort inference algorithm could be extended in a number of ways. Currently, the ACTRESS subset of action notation does not include actions which escape. However, escaping actions are typically used to specify languages with exceptions or exit jumps (e.g. exit- and return-statements in ADA). If the ACTRESS system is to be able to handle languages with exceptions, then the sort inference algorithm will have to be extended to include escaping as a possible outcome. This should be possible. Since an escaping action is only allowed to yield transients (i.e. no bindings), we could extend the action sorts to include a second transient scheme for the transients given if the action escapes, i.e. an action sort would become:

$$A : (\tau, \beta) \hookrightarrow (\tau', \beta', \tau_E')$$

where τ_E' represents the sort of transients produced by the action if it escapes. The natural semantics of ACTRESS action notation could be similarly extended with escaping actions.

Also, the current sort inference algorithm does not include information about the commitment status of an action. As was shown in OASIS[Ørb93], at least partial information can be inferred from a static analysis of the action. Again, this work could be incorporated into our sort inference algorithm, and would provide even greater information about the sort of an action.

Finally, there are related problems in type inference. Recent work has focussed on type inference for dynamically-typed languages [AM91,Tha91,CF91,Hen92,AW93, HR95]. Henglein and Rehof[Hen92, HR95] have addressed the problem of inserting dynamic type checks in SCHEME programs. Significantly, they have identified the minimum number of checks that must be inserted to guarantee the program will run without a type error. Currently the action notation sort checker does not attempt to minimize the number of sort checks inserted. For example, a datum may be checked several times at each different point of use, rather than once at the point of production. The sort checker could, therefore, be improved in this respect. Additionally, the type systems used by Aiken et al [AW93,AWL94] include values as types, a feature that is clearly relevant to action notation.

8.3.2 Sort Inference of Specifications

As we saw in Chapter 2, the action-semantic description of a programming language includes not only clauses for each of the equations in the semantic functions, but also their functionalities. The current actioneer generator however ignores this information. By improving the actioneer generator, it would be possible to perform a sort analysis of the complete action semantic specification. Such an analysis could be used for two purposes:

- **Improved compiler-generation time checks**. Ideally as many errors as possible should be detected at compiler generation time. This would provide timely feedback on the consistency of the language specification, and prevent inconsistent specifications from being used to generate compilers that do not compile, or which only generate errors when used.
- **Improvements in the generated compiler**. In theory, sort information gathered at compiler generation time could be used to improve the quality of

the generated compiler. For example, if the language specification obeyed certain properties, then it may be possible to replace the heavy-weight action notation sort checker with a less complex one. A simpler sort checker would reduce the compile times of the generated compiler.

Additionally, it is hoped that a sort analysis of a semantic specification would allow some properties of the language's type system to be discovered. Doh and Schmidt[DS92a,DS92b] have already studied how to present sort information extracted from an action-semantic specification as a set of typing rules for the language. We would hope to demonstrate that a language was statically-typed, or more precisely, statically "sort-checkable", i.e. that *no* program in the language generates an action that requires run-time sort checks.

In theory, we could develop an improved actioneer generator which used the declared functionalities of the operations in the action semantic description, and an enhanced version of the action notation sort checker to perform sort inference on the description itself. This would allow us to detect certain inconsistencies in the semantic description at compiler-generation time. For example, it would report an error for the incorrect use of "`execute ~D`" mentioned in Section 4.7.

The main difficulty with this approach is the reduced information we would have about the declarative facet. For a particular action, the tokens are known statically, but in a specification, the tokens are unknown as they are represented by syntactic variables. For example, in NANO-Δ, the semantic equation for elaborating a new constant declaration is:

- elaborate [["const" *I*:Identifier "~" *E*:Expression]] =
  evaluate *E* then bind *I* to the value .

Here, the binding is to an unknown token denoted by the syntactic variable *I*, rather than a particular token such as "x". This would prevent us from determining the precise bindings received or produced by an action. We would, however, still know if an action required or produced empty or non-empty bindings. This would still permit some sort errors to be detected.

Consider the following revised sorts for an action A, and a yielder Y:

$$A : (t, b) \hookrightarrow (t', b')$$

$$Y : (t, b) \rightsquigarrow S$$

Here, t and t' are the same as before, but b now represents whether or not an action (or yielder) uses the received bindings, and b' represents whether or not an action produces bindings.

Let $b =$ **yes** if the action definitely does use the received bindings; let $b =$ **no** if the action definitely does not use the received bindings; and let $b =$ **maybe** if the action may or may not use the received bindings. Similarly, let $b' =$ **yes** if the action definitely does produce bindings; let $b' =$ **no** if the action definitely does not produce bindings; and let $b' =$ **maybe** if the action may or may not produce bindings.

It is possible to translate the action sorts used in the functionalities of the semantic functions into this notation. If an action sort specifies some incomes (or outcomes), then the presence of a particular income (or outcome) indicates what an action *may* do. For example a "binding" action may produce bindings, and an action "using current bindings" may access the received bindings. Similarly, if an action specifies some incomes (or outcomes), then the absence of a particular income (or outcome) indicates what an action *does not* do. If an action sort contains no incomes (or outcomes), then the action *may* use any received information (or *may* produce any information). The action sorts in the functionalities for "evaluate", "execute" and "elaborate" in Figure 4.5 are respectively translated as:

- action[giving a value][using current bindings | current storage] is translated to $(\{\ \}, \textbf{maybe}) \hookrightarrow (\{0\text{: value}\}, \textbf{no})$

- action[storing | diverging][using current bindings | current storage] is translated to ({ }, **maybe**) $\hookrightarrow$ ({ }, **no**)
- action[storing | binding][using current bindings | current storage] is translated to ({ }, **maybe**) $\hookrightarrow$ ({ }, **maybe**)

Next, we can calculate the sort of the action on the right-hand side of each of the semantic equations, and compare this sort with the sort of the semantic function. If the two sorts are not consistent, then the semantic equation contains an error. In the case of bindings, an action sort that may use bindings (**maybe**) is consistent with an action sort that does use bindings (**yes**), or with one that does not use bindings (**no**). An action sort that does use bindings (**yes**) is not consistent with an action sort that does not use bindings (**no**).

The sort of an action is calculated by combining the sorts of the primitive actions and yielders it contains. If an action contains an application of a semantic function, then that application is assigned the action sort of the semantic function. The sort of a primitive action is straightforward. For example, the action "bind" *does* produce a binding; the yielder "the_bound to_" *does* access the received bindings; and the action "complete" *does not* use the current bindings, and *does not* produce any bindings.

Using these ideas, we assign the following sorts to the semantic equations given in Figure 4.5:

- `evaluate [[ IDENT ... ]]` : ({ }, **yes**) $\hookrightarrow$ ({0: value}, **no**)
- `execute [[ SEQ ... ]]` : ({ }, **maybe**) $\hookrightarrow$ ({ }, **no**)
- `execute [[ WHILE ... ]]` : ({ }, **maybe**) $\hookrightarrow$ ({ }, **no**)
- `execute [[ LET ... ]]` : ({ }, **maybe**) $\hookrightarrow$ ({ }, **no**)
- `elaborate [[ CONST ... ]]` : ({ }, **maybe**) $\hookrightarrow$ ({ }, **yes**)

Thus we have shown that all of the above semantic equations are consistent with the declared functionality of their corresponding semantic function.

It is also possible to use an analogous approach to classify actions which do, do not, and may access storage or modify storage.

Using these techniques would allow the actioneer generator to detect a wide range of errors in the language specification, and improve the feedback given to the language designer at compiler-generation time. Indeed, the ability to check a specification for errors is a useful tool in the language design process in its own right.

8.3.3 Standard Action Notation

ACTRESS action notation is different from standard action notation for historical reasons. Ideally, the ACTRESS system should be updated to use standard action notation. For sort inference, this would mean inferring tuple sorts for transients, rather than record sorts. We believe that this is possible given only tuple sorts of known length, for example, an action may not produce transients of sort "$\mathsf{integer}^*$". If this is the case, then we believe that tuple sorts and record sorts are isomorphic (note that tuples in Standard ML are actually syntactic sugar for records).

8.3.4 Integration

In general, programming language design is poorly supported by tools. Typically, there are no tools to support the editing and checking of specifications. Recent work by Mosses and van Deursen[vDM94] has produced the Action Semantics Description (ASD) tools. This provides a system for editing and checking the syntax of specifications, and for automatically translating a source program into its corresponding action. The ASD tools are implemented in an algebraic specification system called ASF+SDF[Kli93,HHKR89,BHK89]. Watt[Wat94] has investigated adding an action interpreter to the ASD tools, to provide a means of performing actions. However, the underlying system operates by repeatedly re-writing the action term, and is, therefore, unlikely to provide an efficient means of performing actions.

Ideally, the ACTRESS system could be integrated with the ASD tools to allow an efficient compiler to be generated from the specification at the click of a button. This would provide the first system that matches Pleban's goal of a *language designer's workbench*.

Bibliography

[AM91] A. Aiken and B. R. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 279–290, 1991.

[AMT94] A. W. Appel, J. S. Mattson, and D. R. Tarditi. *A lexical analyser generator for Standard ML, version 1.6.0*, 1994.

[AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Conference on Functional Programming Languages and Computer Architecture '93*, pages 31–41. ACM, 1993.

[AWL94] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[BBK+82] J. M. Bodwin, L. Bradley, K. Kanda, D. Little, 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.

[BHK89] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM Press and Addison-Wesley, 1989.

[BMW92a] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator (summary). In R. Heldal, C. K. Holst, and P. L. Wadler, editors, *Functional Programming, Glasgow 1991*, BCS Workshops in Computing. Springer-Verlag, 1992.

[BMW92b] D. F. Brown, H. Moura, and D. A. Watt. ACTRESS: an action semantics directed compiler generator. In *Compiler Construction '92*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[Bon91] A. Bondorf. Automatic autoprojection of higher-order recursive equations. *Science of Computer Programming*, 16:151–195, 1991.

[Bon92] A. Bondorf. Improving bindings times without explicit cps-conversion. In *ACM Conference on Lisp and Functional Programming*, pages 1–10, 1992.

[BP93] A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 308–317, 1993.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[CM89] L. Cardelli and J. C. Mitchell. Operations on records. In *Workshop on Mathematical Foundations of Programming Language Semantics*, Lecture Notes in Computer Science. Springer, 1989.

[Con88] C. Consel. New insights into partial evaluation: the Schism experiment. In H. Ganzinger, editor, *2nd European Symposium on Programming (ESOP'88)*, volume 300 of *Lecture Notes in Computer Science*, pages 236–246. Springer-Verlag, 1988.

[Con93] C. Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1993.

[Doh95] K.-G. Doh. Action transformation by partial evaluation. In *PEPM'95, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, June 1995.

[DS92a] K.-G. Doh and D. A. Schmidt. Action semantics-directed prototyping. Report 92-30, Computing and Info. Science Dept., Kansas State Univ., 1992.

[DS92b] K.-G. Doh and D. A. Schmidt. Extraction of strong typing laws from action semantics definitions. In *4th European Symposium on Programming (ESOP'92)*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[ES90] S. Even and D. A. Schmidt. Type inference for action semantics. In *3rd European Symposium on Programming (ESOP'90)*, Lecture Notes in Computer Science No 432, pages 118–133. Springer-Verlag, Berlin, 1990.

[ESL89] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG - A generator for efficient back ends. *ACM SIGPLAN Notices*, 24(7):227–237, July 1989.

[GE90] J. Grosch and H. Emmelmann. A tool box for compiler construction. In *Compiler Construction '90, Schwerin, FRG*, Lecture Notes in Computer Science, pages 106–116. Springer-Verlag, October 1990.

[GHL+92] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: a complete, flexible compiler construction system. *Communications of the ACM*, pages 121–130, February 1992.

[GJ91] 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.

[Gla96] Glasgow Functional Programming Group. *Glasgow Haskell Compiler, version 2.01*, 1996.

[Gue81] I. Guessarian. *Algebraic Semantics*, volume 99 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[Hen92] F. Henglein. Dynamic typing. In B. Krieg-Brückner, editor, *4th European Symposium on Programming (ESOP'92)*, pages 233–253. Springer-Verlag, Feb 1992. Lecture Notes in Computer Science, Vol. 582.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDFemdash reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, November 1989.

[HR95] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture (FPCA), La Jolla, California*. ACM, ACM Press, June 1995.

[Joh75] S. C. Johnson. YACC: yet another compiler compiler. C.S. Technical Report, 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[Kas93] U. Kastens. Attribute grammars in a compiler construction environment. In H. Alblas and B. Melichar, editors, *Attribute grammars applications and systems*, pages 380–400. Springer Verlag, 1993. Lecture Notes in Computer Science 545.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, March 1993.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–164, March 1966.

[Lee89] P. Lee. *Realistic Compiler Generation*. Foundations of Computing. MIT Press, Cambridge, Massachusetts, 1989.

[LH96] S. Liang and P. Hudak. Modular denotational semantics for compiler construction. In *6th European Symposium on Programming (ESOP'96)*, Lecture Notes in Computer Science No 432. Springer-Verlag, Berlin, 1996.

[LP87] P. Lee and U. F. Pleban. A realistic compiler generator system based on high-level semantics. In *14th ACM Symposium on Principles of Programming Languages*, pages 284–295. ACM, 1987.

[LS75] M. E. Lesk and E. Schmidt. LEX: a lexical analyzer generator. In *UNIX Programmer's Manual 2*, Murray Hill, NJ, 1975. AT&T Bell Laboratories.

[Mei92] E. Meijer. *Calculating Compilers*. PhD thesis, Department of Computer Science, Nijmegen University, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17,3*, pages 348–375, 1978.

[Mog90] E. Moggi. An abstract view of programming languages. ECS-LFCS-90-113, Laboratory for the Foundations of Computer Science, Computer Science, University of Edinburgh, Edinburgh, Scotland, 1990.

[Mos79] P. D. Mosses. SIS — semantics implementation system. DAIMI MD-30, Computer Science Department, Aarhus University, Aarhus, Denmark, 1979.

[Mos92] P. D. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[Mou93a] H. Moura. *Action transformations.* PhD thesis, Department of Computing Science, Glasgow University, 1993.

[Mou93b] H. Moura. An implementation of action semantics. Report FM-1993-??, Department of Computing Science, Glasgow University, 1993.

[NN92] F. Neilsen and H. R. Nielsen. *Two-level Functional Languages.* Cambridge Tracts in Theoretical Computer Science 34. Cambridge University Press, 1992.

[Ørb93] P. Ørbæk. Analysis and optimization of actions. M.Sc. dissertation, Computer Science Department, Aarhus University, Denmark, September 1993.

[Ørb94] P. Ørbæk. OASIS: An optimizing action-based compiler generator. In Peter Fritzon, editor, *Proceedings of the 1994 Conference on Compiler Construction, Edinburgh,* volume 786 of LNCS, pages 1–15. Springer-Verlag, April 1994.

[Pal92a] 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.

[Pal92b] J. Palsberg. A provably correct compiler generator. In *4th European Symposium on Programming (ESOP'92),* volume 582 of *Lecture Notes in Computer Science,* pages 418–434. Springer-Verlag, 1992.

[Pau81] L. W. Paulson. A compiler generator for semantic grammars. STAN-CS-82-893, Stanford University, 1981.

[PF92] M. Pettersson and P. Fritzon. DML - a meta-language and system for the generation of practical and efficient compilers from denotational specifications. In *International Conference on Computer Languages,* pages 127–136. IEEE, April 1992.

[Sch86] D. A. Schmidt. *Denotational Semantics — A methodology for language development.* Allyn Bacon, Newton, Massachusetts, 1986.

[Sch91] D. A. Schmidt, April 1991. Personal Correspondence.

[Sch94] D. A. Schmidt. *The Structure of Typed Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1994.

[Sto77] J. E. Stoy. *The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[TA90] D. R. Tarditi and A. W. Appel. *ML-Yacc, version 2.0,* 1990. Documentation for Release Version.

[Tha91] S. R. Thatte. Quasi-static typing. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 367–381, 1991.

[vDM94] A. van Deursen and P. Mosses. A demonstration of ASD, the action semantics description tools. In *Proceedings of the First International Workshop on Action Semantics*, pages 56–59, 1994.

[Wad90] P. L. Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Symposium on Lisp and Functional Programming*, 1990.

[Wai93] W. Waite. An executable language definition. *ACM SIGPLAN Notices*, 28:21–40, February 1993.

[Wan82] M. Wand. Deriving target code as a representation of continuation semantics. In *ACM Transactions on Programming Languages and Systems*, pages 496–517. ACM, July 1982.

[Wan84] M. Wand. A semantic prototyping system. In *Proceedings of the SIGPLAN'84 Symposium on Compiler Construction*, SIGPLAN Notices 19,6, pages 213–221. ACM, 1984.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, 1987.

[Wan89] M. Wand. Type inference for record concatenation and simple objects. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 92–97, 1989.

[Wat90] D. A. Watt. *Programming Language Concepts and Paradigms.* Prentice-Hall International, Hemel Hempstead, England, 1990.

[Wat91] D. A. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall International, Hemel Hempstead, England, 1991.

[Wat93] D. A. Watt. *Programming Language Processors.* Prentice-Hall International, Hemel Hempstead, England, 1993.

[Wat94] D. A. Watt. Using ASF+SDF to interpret and transform actions. In *Proceedings of the First International Workshop on Action Semantics*, pages 129–142, 1994.

[Wat96] D. A. Watt. Why don't programming language designers use formal methods? In R. Barros, editor, *SEMISH '96*, Federal University of Pernambuco, Brazil, 1996.

[WS91] L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., 1991.

Appendix A

Formal Summary of Action Semantics

A.1 Abstract Syntax

Syntactic variables

- A: Action
- Y: Yielder
- S: Sort
- n: natural
- k: token
- C is a data constant
- O is a data operation

Production rules

A	::=	complete | fail | A_1 or A_2 | A_1 and A_2 | A_1 and then A_2 | unfolding A | give Y label # n | give Y | check Y | A_1 then A_2 | bind k to Y | furthermore A | A_1 hence A_2 | A_1 moreover A_2 | A_1 before A_2 | store Y_1 in Y_2 | deallocate Y | enact Y | A_1 else A_2 | recursively bind k to Y | allocate S
Y	::=	C | O $(Y_1, \ldots, Y_n)$ | the S # n | the S | it | Y_1 is Y_2 | if Y_1 then Y_2 else Y_3 | the S bound to k | the S stored in Y | abstraction A | Y_1 with Y_2 | closure Y
S	::=	truth-value | integer | list[of S] | cell[of S] | abstraction | action

A.2 Semantics

Semantic variables

- A: action
- Y: yielder[of a datum]
- $S \leq$ datum
- d: datum
- c: cell
- n: natural
- k: token
- t: transients = map[of natural to datum]
- b: bindings = map[of token to datum]
- s: storage = map[cell to (datum | uninitialized)]
- o: $\{completed, diverged, failed\}$

Notation

- $(t, b, s) \vdash A \Rightarrow (o', t', b', s')$ means that a performance of action A, with income (t, b, s), can result in the outcome (o', t', b', s'). If $o' = failed$, $t' = b' = \{ \}$.
- $(t, b, s) \vdash Y \Rightarrow d$ means that an evaluation of yielder Y, with income (t, b, s), will yield datum d.
- *dom m* means the domain of the map *m*.
- *mergeable m m′* means that maps *m* and *m′* have disjoint domains, i.e., that $dom\ m \cap dom\ m' = \{ \}$.
- *merge m m′* means the map obtained by merging maps *m* and *m′* (defined only if *m* and *m′* have disjoint domains).
- *overlay m m′* means the map obtained by overlaying map *m* on to map *m′*.

- *modify* x y m means the map obtained by perturbing map m such that x maps to y.
- *remove* x m means the map obtained by removing x from the domain of map m.

Conventions

- For any action A and income (t, b, s), if no inference rule specifies otherwise, then $(t, b, s) \vdash A \Rightarrow (\mathit{failed}, \{\ \}, \{\ \}, s')$.
- For any yielder Y and income (t, b, s), if no inference rule specifies otherwise, then $(t, b, s) \vdash Y \Rightarrow \textsf{nothing}$.

A.2.1 Basic Action Notation

(COMPLETE-S1)
$$\frac{}{(t, b, s) \vdash \textsf{complete} \Rightarrow (\mathit{completed}, \{\ \}, \{\ \}, s)}$$

(FAIL-S1)
$$\frac{}{(t, b, s) \vdash \textsf{fail} \Rightarrow (\mathit{failed}, \{\ \}, \{\ \}, s)}$$

(OR-S1)
$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{failed}, \{\ \}, \{\ \}, s_1)\ ;\ (t, b, s) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)}{(t, b, s) \vdash A_1 \textsf{ or } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(OR-S2)
$$\frac{(t, b, s) \vdash A_2 \Rightarrow (\mathit{failed}, \{\ \}, \{\ \}, s_2)\ ;\ (t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)}{(t, b, s) \vdash A_1 \textsf{ or } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(OR-S3)
$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)\ ;\ o_1 \neq \mathit{failed}}{(t, b, s) \vdash A_1 \textsf{ or } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(OR-S4)
$$\frac{(t, b, s) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)\ ;\ o_2 \neq \mathit{failed}}{(t, b, s) \vdash A_1 \textsf{ or } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(AND-S1)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ; \quad (t, b, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)\ ; \quad \mathit{mergeable}\ t_1\ t_2;\ \mathit{mergeable}\ b_1\ b_2}{(t, b, s) \vdash A_1 \text{ and } A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, \mathit{merge}\ b_1\ b_2, s_2)}$$

(AND-S2)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)\ ;\ o_1 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ and } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(AND-S3)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ; \quad (t, b, s_1) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)\ ;\ o_2 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ and } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(AND-THEN-S1)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ; \quad (t, b, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)\ ; \quad \mathit{mergeable}\ t_1\ t_2;\ \mathit{mergeable}\ b_1\ b_2}{(t, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, \mathit{merge}\ b_1\ b_2, s_2)}$$

(AND-THEN-S2)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)\ ;\ o_1 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(AND-THEN-S3)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ; \quad (t, b, s_1) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)\ ;\ o_2 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ and then } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(UNFOLDING-S1)

$$\frac{(t, b, s) \vdash \text{A[unfold / unfolding } A] \Rightarrow (o', t', b', s')}{(t, b, s) \vdash \text{unfolding } A \Rightarrow (o', t', b', s')}$$

(CONSTANT-S1)

$$\frac{C\colon S}{(t, b, s) \vdash C \Rightarrow C}$$

(OPERATION-S1)

$$\frac{\begin{array}{c} O\colon S_1 \times \ldots \times S_n \to S\ ; \\ (t, b, s) \vdash Y_1 \Rightarrow d_1\ ;\ \ldots\ ;\ (t, b, s) \vdash Y_n \Rightarrow d_n\ ; \\ d_1\colon S_1\ ;\ \ldots\ ;\ d_n\colon S_n \end{array}}{(t, b, s) \vdash O(Y_1, \ldots, Y_n) \Rightarrow O(d_1, \ldots, d_n)}$$

Note

unfolding A = A[unfold / unfolding A]

A.2.2 Functional Action Notation

(GIVE-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow d}{(t, b, s) \vdash \textsf{give } Y \textsf{ label } \#n \Rightarrow (\textit{completed}, \{n \mapsto d\}, \{\ \}, s)}$$

(GIVE-S2)

$$\frac{(t, b, s) \vdash Y \Rightarrow d}{(t, b, s) \vdash \textsf{give } Y \Rightarrow (\textit{completed}, \{0 \mapsto d\}, \{\ \}, s)}$$

(CHECK-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow \textsf{true}}{(t, b, s) \vdash \textsf{check } Y \Rightarrow (\textit{completed}, \{\ \}, \{\ \}, s)}$$

(CHECK-S2)

$$\frac{(t, b, s) \vdash Y \Rightarrow \textsf{false}}{(t, b, s) \vdash \textsf{check } Y \Rightarrow (\textit{failed}, \{\ \}, \{\ \}, s)}$$

(THEN-S1)

$$\frac{\begin{array}{c} (t, b, s) \vdash A_1 \Rightarrow (\textit{completed}, t_1, b_1, s_1)\ ; \\ (t_1, b, s_1) \vdash A_2 \Rightarrow (\textit{completed}, t_2, b_2, s_2)\ ;\ \textit{mergeable } b_1\ b_2 \end{array}}{(t, b, s) \vdash A_1 \textsf{ then } A_2 \Rightarrow (\textit{completed}, t_2, \textit{merge } b_1\ b_2, s_2)}$$

(THEN-S2)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)\ ;\ o_1 \neq \textit{completed}}{(t, b, s) \vdash A_1 \textsf{ then } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(THEN-S3)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ;\quad (t_1, b, s_1) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)\ ;\ o_2 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ then } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(THE-S1)

$$\frac{n \in \mathit{dom}\, t\ ;\ t(n){:}\ S}{(t, b, s) \vdash \text{the } S \text{ \#}n \Rightarrow t(n)}$$

(THE-S2)

$$\frac{d{:}\ S}{(\{0 \mapsto d\}, b, s) \vdash \text{the } S \Rightarrow d}$$

(IT-S1)

$$\frac{}{(\{0 \mapsto d\}, b, s) \vdash \text{it} \Rightarrow d}$$

(IS-S1)

$$\frac{(t, b, s) \vdash Y_1 \Rightarrow d_1\ ;\ (t, b, s) \vdash Y_2 \Rightarrow d_2\ ;\ d_1 = d_2}{(t, b, s) \vdash Y_1 \text{ is } Y_2 \Rightarrow \text{true}}$$

(IS-S2)

$$\frac{(t, b, s) \vdash Y_1 \Rightarrow d_1\ ;\ (t, b, s) \vdash Y_2 \Rightarrow d_2\ ;\ d_1 \neq d_2}{(t, b, s) \vdash Y_1 \text{ is } Y_2 \Rightarrow \text{false}}$$

(IF-S1)

$$\frac{(t, b, s) \vdash Y_1 \Rightarrow \text{true}\ ;\ (t, b, s) \vdash Y_2 \Rightarrow d_2}{(t, b, s) \vdash \text{if } Y_1 \text{ then } Y_2 \text{ else } Y_3 \Rightarrow d_2}$$

(IF-S2)

$$\frac{(t, b, s) \vdash Y_1 \Rightarrow \text{false}\ ;\ (t, b, s) \vdash Y_3 \Rightarrow d_3}{(t, b, s) \vdash \text{if } Y_1 \text{ then } Y_2 \text{ else } Y_3 \Rightarrow d_3}$$

Notes

it = the datum

give Y = give Y label #0

A.2.3 Declarative Action Notation

(BIND-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow d \;;\; d\text{: bindable}}{(t, b, s) \vdash \text{bind } k \text{ to } Y \Rightarrow (\mathit{completed}, \{\ \}, \{k \mapsto d\}, s)}$$

(FURTHERMORE-S1)

$$\frac{(t, b, s) \vdash A \Rightarrow (\mathit{completed}, t', b', s')}{(t, b, s) \vdash \text{furthermore } A \Rightarrow (\mathit{completed}, t', \mathit{overlay}\ b'\ b, s')}$$

(FURTHERMORE-S2)

$$\frac{(t, b, s) \vdash A \Rightarrow (o', t', b', s') \;;\; o' \neq \mathit{completed}}{(t, b, s) \vdash \text{furthermore } A \Rightarrow (o', t', b', s')}$$

(HENCE-S1)

$$\frac{\begin{array}{c}(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1) \;; \\ (t, b_1, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2) \;;\; \mathit{mergeable}\ t_1\ t_2\end{array}}{(t, b, s) \vdash A_1 \text{ hence } A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, b_2, s_2)}$$

(HENCE-S2)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1) \;;\; o_1 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ hence } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(HENCE-S3)

$$\frac{\begin{array}{c}(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1) \;; \\ (t, b_1, s_1) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2) \;;\; o_2 \neq \mathit{completed}\end{array}}{(t, b, s) \vdash A_1 \text{ hence } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(MOREOVER-S1)

$$\frac{\begin{array}{c}(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1) \;; \\ (t, b, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2) \;;\; \mathit{mergeable}\ t_1\ t_2\end{array}}{\begin{array}{r}(t, b, s) \vdash A_1 \text{ moreover } A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, \\ \mathit{overlay}\ b_2\ b_1, s_2)\end{array}}$$

(MOREOVER-S2)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1) \;;\; o_1 \neq \mathit{completed}}{(t, b, s) \vdash A_1 \text{ moreover } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(MOREOVER-S3)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ;\quad (t, b, s_1) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)\ ;\ o_2 \neq \mathit{completed}}{(t, b, s) \vdash A_1\ \textsf{moreover}\ A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(BEFORE-S1)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ;\quad (t, \mathit{overlay}\ b_1\ b, s_1) \vdash A_2 \Rightarrow (\mathit{completed}, t_2, b_2, s_2)\ ;\ \mathit{mergeable}\ t_1\ t_2}{(t, b, s) \vdash A_1\ \textsf{before}\ A_2 \Rightarrow (\mathit{completed}, \mathit{merge}\ t_1\ t_2, \mathit{overlay}\ b_2\ b_1, s_2)}$$

(BEFORE-S2)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)\ ;\ o_1 \neq \mathit{completed}}{(t, b, s) \vdash A_1\ \textsf{before}\ A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(BEFORE-S3)

$$\frac{(t, b, s) \vdash A_1 \Rightarrow (\mathit{completed}, t_1, b_1, s_1)\ ;\quad (t, \mathit{overlay}\ b_1\ b, s_1) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)\ ;\ o_2 \neq \mathit{completed}}{(t, b, s) \vdash A_1\ \textsf{before}\ A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(BOUND-S1)

$$\frac{k \in \mathit{dom}\ b\ ;\ b(k){:}\ S}{(t, b, s) \vdash \textsf{the}\ S\ \textsf{bound to}\ k \Rightarrow b(k)}$$

Note

furthermore A = rebind moreover A

A.2.4 Imperative Action Notation

(STORE-S1)

$$\frac{(t, b, s) \vdash Y_1 \Rightarrow d\ ;\ (t, b, s) \vdash Y_2 \Rightarrow c\ ;\ c{:}\ \mathrm{cell}[S]\ ;\ d{:}\ S}{(t, b, s) \vdash \textsf{store}\ Y_1\ \textsf{in}\ Y_2 \Rightarrow (\mathit{completed}, \{\ \}, \{\ \}, \mathit{modify}\ c\ d\ s)}$$

(DEALLOCATE-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow c\ ;\ c \in \mathit{dom}\ s}{(t, b, s) \vdash \textsf{deallocate}\ Y \Rightarrow (\mathit{completed}, \{\ \}, \{\ \}, \mathit{remove}\ c\ s)}$$

(STORED-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow c\ ;\ c \in \mathit{dom}\ s\ ;\ s(c){:}\ S}{(t, b, s) \vdash \textsf{the}\ S\ \textsf{stored in}\ Y \Rightarrow s(c)}$$

A.2.5 Reflective Action Notation

(ENACT-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow \mathit{abstraction}(A, t_0, b_0)\ ;\ (t_0, b_0, s) \vdash A \Rightarrow (o', t', b', s')}{(t, b, s) \vdash \text{enact } Y \Rightarrow (o', t', b', s')}$$

(ABSTRACTION-S1)

$$\frac{}{(t, b, s) \vdash \text{abstraction } A \Rightarrow \mathit{abstraction}(A, \{\ \}, \{\ \})}$$

(WITH-S1)

$$\frac{(t, b, s) \vdash Y_1 \Rightarrow \mathit{abstraction}(A, \{\ \}, b_0)\ ;\ (t, b, s) \vdash Y_2 \Rightarrow d}{(t, b, s) \vdash Y_1 \text{ with } Y_2 \Rightarrow \mathit{abstraction}(A, \{0 \mapsto d\}, b_0)}$$

(CLOSURE-S1)

$$\frac{(t, b, s) \vdash Y \Rightarrow \mathit{abstraction}(A, t_0, \{\ \})}{(t, b, s) \vdash \text{closure } Y \Rightarrow \mathit{abstraction}(A, t_0, b)}$$

A.2.6 Hybrid Action Notation

(ELSE-S1)

$$\frac{(\{\ \}, b, s) \vdash A_1 \Rightarrow (o_1, t_1, b_1, s_1)}{(\{0 \mapsto \text{true}\}, b, s) \vdash A_1 \text{ else } A_2 \Rightarrow (o_1, t_1, b_1, s_1)}$$

(ELSE-S2)

$$\frac{(\{\ \}, b, s) \vdash A_2 \Rightarrow (o_2, t_2, b_2, s_2)}{(\{0 \mapsto \text{false}\}, b, s) \vdash A_1 \text{ else } A_2 \Rightarrow (o_2, t_2, b_2, s_2)}$$

(REC-BIND-S1)

$$\frac{(t, \mathit{overlay}\ \{k \mapsto d\}\ b, s) \vdash Y \Rightarrow d\ ;\ d\text{: bindable}}{(t, b, s) \vdash \text{recursively bind } k \text{ to } Y \Rightarrow (\mathit{completed}, \{\ \}, \{k \mapsto d\}, s)}$$

(ALLOCATE-S1)

$$\frac{c\text{: } S \leq \text{cell}\ ;\ c \notin \mathit{dom}\ s}{\begin{array}{l}(t, b, s) \vdash \text{allocate } S \Rightarrow (\mathit{completed}, \{0 \mapsto c\}, \{\ \}, \\ \qquad\qquad\qquad \mathit{modify}\ c \text{ uninitialized } s)\end{array}}$$

Note

A_1 else A_2 = (check (it is true) then A_1) or (check (it is false) then A_2)

Appendix B

Sort Inference Rules

B.1 Notation

B.1.1 Variable Naming Conventions

ε: an environment mapping symbols to sorts.

S, S_1, S_2: sort terms.

Y, Y_1, Y_2: yielder terms.

A, A_1, A_2: action terms.

τ, τ_1, τ_1', τ_A: record sort schemes for transients.

β, β_1, β_1', β_A: record sort schemes for bindings.

θ: a sort variable.

ρ: a record (row) variable.

σ, σ', σ_1, σ_2: a data sort scheme.

B.2 Basic Action Notation

(COMPLETE-I)

$$\frac{}{\varepsilon \vdash \text{complete}: (\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{\ \}, \{\ \})}$$

(FAIL-I)

$$\frac{}{\varepsilon \vdash \text{fail}: \text{nothing}}$$

(AND-I)

$$\frac{\varepsilon \vdash A_1: (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1')\ ;\ \varepsilon \vdash A_2: (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{c}\varepsilon \vdash A_1 \text{ and } A_2: (\textit{distribute}\ \tau_1\ \tau_2, \textit{distribute}\ \beta_1\ \beta_2) \hookrightarrow \\ (\textit{merge}\ \tau_1'\ \tau_2', \textit{merge}\ \beta_1'\ \beta_2')\end{array}}$$

(AND-THEN-I)

$$\frac{\varepsilon \vdash A_1: (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1')\ ;\ \varepsilon \vdash A_2: (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{c}\varepsilon \vdash A_1 \text{ and then } A_2: (\textit{distribute}\ \tau_1\ \tau_2, \textit{distribute}\ \beta_1\ \beta_2) \hookrightarrow \\ (\textit{merge}\ \tau_1'\ \tau_2', \textit{merge}\ \beta_1'\ \beta_2')\end{array}}$$

(OR-I)

$$\frac{\varepsilon \vdash A_1: (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1')\ ;\ \varepsilon \vdash A_2: (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{c}\varepsilon \vdash A_1 \text{ or } A_2: (\textit{switch}\ \tau_1\ \tau_2, \textit{switch}\ \beta_1\ \beta_2) \hookrightarrow \\ (\textit{select}\ \tau_1'\ \tau_2', \textit{select}\ \beta_1'\ \beta_2')\end{array}}$$

(ELSE-I)

$$\frac{\begin{array}{c}\varepsilon \vdash A_1: (\{0\text{: truth-value}\}\gamma, \beta_1) \hookrightarrow (\{\ \}, \beta_1')\ ; \\ \varepsilon \vdash A_2: (\{0\text{: truth-value}\}\gamma, \beta_2) \hookrightarrow (\{\ \}, \beta_2')\end{array}}{\begin{array}{c}\varepsilon \vdash A_1 \text{ else } A_2: (\{0\text{: truth-value}\}\gamma, \textit{switch}\ \beta_1\ \beta_2) \hookrightarrow \\ (\{\ \}, \textit{select}\ \beta_1'\ \beta_2')\end{array}}$$

(UNFOLDING-I)

$$\frac{[\text{unfold}: (\tau, \beta) \hookrightarrow (\tau'\ \beta')]\ \varepsilon \vdash A: (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \text{unfolding } A: (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

(UNFOLD-I)

$$\frac{}{[\text{unfold}: (\tau, \beta) \hookrightarrow (\tau'\ \beta')]\ \varepsilon \vdash \text{unfold}: (\tau, \beta) \hookrightarrow (\tau'\ \beta')}$$

(IS-I)

$$\frac{\varepsilon \vdash Y_1: (\tau_1, \beta_1) \rightsquigarrow \sigma\ ;\ \ \varepsilon \vdash Y_2: (\tau_2, \beta_2) \rightsquigarrow \sigma}{\varepsilon \vdash Y_1 \text{ is } Y_2: (\textit{distribute}\ \tau_1\ \tau_2, \textit{distribute}\ \beta_1\ \beta_2) \rightsquigarrow \text{truth-value}}$$

(AN-I)

$$\frac{\varepsilon \vdash S\colon \sigma}{\varepsilon \vdash \textsf{an}\ S\colon \sigma}$$

(SORT-NAME-I)

$$\frac{}{[S\colon \sigma]\ \varepsilon \vdash S\colon \sigma}$$

(JOIN-I)

$$\frac{\varepsilon \vdash S_1\colon \sigma_1\ ;\ \ \varepsilon \vdash S_2\colon \sigma_2}{\varepsilon \vdash S_1\ !\ S_2\colon \sigma_1 \mid \sigma_2}$$

(LIST-I)

$$\frac{\varepsilon \vdash S\colon \sigma}{\varepsilon \vdash \textsf{list}\ [\,S\,]\colon \text{list}\ [\sigma]}$$

(YIELDER-I)

$$\frac{\varepsilon \vdash S\colon \sigma}{\varepsilon \vdash S\colon (\{\ \}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \sigma}$$

B.3 Functional Action Notation

(GIVE-I)

$$\frac{\varepsilon \vdash Y\colon (\tau, \beta) \rightsquigarrow \sigma}{\varepsilon \vdash \textsf{give}\ Y\ \textsf{label}\ \#\ n\colon (\tau, \beta) \hookrightarrow (\{n\colon \sigma\}, \{\ \})}$$

(CHECK-I)

$$\frac{\varepsilon \vdash Y\colon (\tau, \beta) \rightsquigarrow \sigma\ ;\ \ \sigma\ \&\ \text{truth-value} \neq \text{nothing}}{\varepsilon \vdash \textsf{check}\ Y\colon (\tau, \beta) \hookrightarrow (\{\ \}, \{\ \})}$$

(THEN-I)

$$\frac{\varepsilon \vdash A_1\colon (\tau_1, \beta_1) \hookrightarrow (\tau, \beta_1')\ ;\ \ \varepsilon \vdash A_2\colon (\tau, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash A_1\ \textsf{then}\ A_2\colon (\tau_1, \mathit{distribute}\ \beta_1\ \beta_2) \hookrightarrow (\tau_2', \mathit{merge}\ \beta_1'\ \beta_2')}$$

(THE-I)

$$\frac{\varepsilon \vdash S\colon \sigma\ ;\ \ \theta\ \&\ \sigma \neq \text{nothing}}{\varepsilon \vdash \textsf{the}\ S\ \#\ n\colon (\{n\colon \theta\}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \theta\ \&\ \sigma}$$

(IT-I)

$$\frac{}{\varepsilon \vdash \textsf{it}\colon (\{0\colon \theta\}\gamma_1, \{\ \}\gamma_2) \rightsquigarrow \theta}$$

B.4 Declarative Action Notation

(BIND-I)

$$\frac{\varepsilon \vdash Y\text{: } (\tau, \beta) \rightsquigarrow \sigma \text{ ; bindable \& } \sigma \neq \text{nothing}}{\varepsilon \vdash \textsf{bind } k \textsf{ to } Y\text{: } (\tau, \beta) \hookrightarrow (\{\ \}, \{k\text{: } \sigma\})}$$

(REC-BIND-I)

$$\frac{\varepsilon \vdash Y\text{: } (\tau, \textit{overlay } \{k\text{: } \sigma\}\ \beta) \rightsquigarrow \sigma \text{ ; bindable \& } \sigma \neq \text{nothing}}{\varepsilon \vdash \textsf{recursively bind } k \textsf{ to } Y\text{: } (\tau, \beta) \hookrightarrow (\{\ \}, \{k\text{: } \sigma\})}$$

(HENCE-I)

$$\frac{\varepsilon \vdash A_1\text{: } (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta) \text{ ; } \varepsilon \vdash A_2\text{: } (\tau_2, \beta) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash A_1 \textsf{ hence } A_2\text{: } (\textit{distribute } \tau_1\ \tau_2, \beta_1) \hookrightarrow (\textit{merge } \tau_1'\ \tau_2', \beta_2')}$$

(MOREOVER-I)

$$\frac{\varepsilon \vdash A_1\text{: } (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1') \text{ ; } \varepsilon \vdash A_2\text{: } (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\begin{array}{l}\varepsilon \vdash A_1 \textsf{ moreover } A_2\text{: } (\textit{distribute } \tau_1\ \tau_2, \textit{distribute } \beta_1\ \beta_2) \hookrightarrow \\ \qquad\qquad (\textit{merge } \tau_1'\ \tau_2', \textit{overlay } \beta_2'\ \beta_1')\end{array}}$$

(FURTHERMORE-I)

$$\frac{\varepsilon \vdash A\text{: } (\tau, \beta) \hookrightarrow (\tau', \beta')}{\begin{array}{l}\varepsilon \vdash \textsf{furthermore } A\text{: } (\tau, \textit{distribute } \{\ \}\rho\ \beta) \hookrightarrow \\ \qquad\qquad (\tau', \textit{overlay } \beta'\ \{\ \}\rho)\end{array}}$$

(BEFORE-I)

$$\frac{\begin{array}{c}\varepsilon \vdash A_1\text{: } (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1') \text{ ;} \\ \varepsilon \vdash A_2\text{: } (\tau_2, \textit{overlay } \beta_1'\ \{\ \}\rho) \hookrightarrow (\tau_2', \beta_2')\end{array}}{\begin{array}{l}\varepsilon \vdash A_1 \textsf{ before } A_2\text{: } (\textit{distribute } \tau_1\ \tau_2, \textit{distribute } \{\ \}\rho\ \beta_1) \hookrightarrow \\ \qquad\qquad (\textit{merge } \tau_1'\ \tau_2', \textit{overlay } \beta_2'\ \beta_1')\end{array}}$$

(BOUND-I)

$$\frac{\varepsilon \vdash S\text{: } \sigma \text{ ; } \theta \text{ \& } \sigma \neq \text{nothing}}{\varepsilon \vdash \textsf{the } S \textsf{ bound to } k\text{: } (\{\ \}\gamma_1, \{k\text{: } \theta\}\gamma_2) \rightsquigarrow \theta \text{ \& } \sigma}$$

B.5 Imperative Action Notation

(ALLOCATE-I)

$$\frac{\varepsilon \vdash S\text{: cell } [\sigma]}{\varepsilon \vdash \textsf{allocate } S\text{: } (\{\ \}\gamma_1, \{\ \}\gamma_2) \hookrightarrow (\{0\text{: cell } [\sigma]\}, \{\ \})}$$

(DEALLOCATE-I)

$$\frac{\varepsilon \vdash Y\text{: } (\tau, \beta) \rightsquigarrow \text{cell } [\sigma]}{\varepsilon \vdash \textsf{deallocate } Y\text{: } (\tau, \beta) \hookrightarrow (\{\ \}, \{\ \})}$$

(STORE-I)

$$\frac{\begin{array}{c}\varepsilon \vdash Y_1\text{: } (\tau_1, \beta_1) \rightsquigarrow \sigma_1 \text{ ;} \\ \varepsilon \vdash Y_2\text{: } (\tau_2, \beta_2) \rightsquigarrow \text{cell } [\sigma_2] \text{ ; } \sigma_1 \mathbin{\&} \sigma_2 \neq \text{nothing}\end{array}}{\varepsilon \vdash \textsf{store } Y_1 \textsf{ in } Y_2\text{: } (\textit{distribute } \tau_1\ \tau_2, \textit{distribute } \beta_1\ \beta_2) \hookrightarrow (\{\ \}, \{\ \})}$$

(STORED-I)

$$\frac{\begin{array}{c}\varepsilon \vdash S_1\text{: } \sigma_1 \text{ ; } \varepsilon \vdash Y_2\text{: } (\tau_2, \beta_2) \rightsquigarrow \text{cell } [\sigma_2] \text{ ;} \\ \sigma_1 \mathbin{\&} \sigma_2 \neq \text{nothing}\end{array}}{\varepsilon \vdash \textsf{the } S_1 \textsf{ stored in } Y_2\text{: } (\tau_2, \beta_2) \rightsquigarrow \sigma_1 \mathbin{\&} \sigma_2}$$

(CELL-I)

$$\frac{\varepsilon \vdash S\text{: } \sigma}{\varepsilon \vdash \textsf{cell } [\, S\,]\text{: cell } [\sigma]}$$

B.6 Reflective Action Notation

(ENACT-I)

$$\frac{\varepsilon \vdash Y\text{: } (\tau, \beta) \rightsquigarrow (\text{abstraction } (\{\ \}, \{\ \}) \hookrightarrow (\tau'_A, \beta'_A))}{\varepsilon \vdash \textsf{enact } Y\text{: } (\tau, \beta) \hookrightarrow (\tau'_A, \beta'_A)}$$

(WITH-I)

$$\frac{\begin{array}{c}\varepsilon \vdash Y_1\text{: } (\tau_1, \beta_1) \rightsquigarrow (\text{abstraction } (\{0\text{: } \sigma'\}, \beta_A) \hookrightarrow (\tau'_A, \beta'_A)) \text{ ;} \\ \varepsilon \vdash Y_2\text{: } (\tau_2, \beta_2) \rightsquigarrow \sigma \text{ ; } \sigma' \mathbin{\&} \sigma \neq \text{nothing}\end{array}}{\begin{array}{l}\varepsilon \vdash Y_1 \textsf{ with } Y_2\text{: } (\textit{distribute } \tau_1\ \tau_2, \textit{distribute } \beta_1\ \beta_2) \rightsquigarrow \\ \qquad\qquad (\text{abstraction } (\{\ \}, \beta_A) \hookrightarrow (\tau'_A, \beta'_A))\end{array}}$$

(CLOSURE-I)

$$\frac{\varepsilon \vdash Y\text{: } (\tau, \beta) \rightsquigarrow (\text{abstraction } (\tau_A, \beta_A) \hookrightarrow (\tau'_A, \beta'_A))}{\begin{array}{l}\varepsilon \vdash \textsf{closure } Y\text{: } (\tau, \textit{distribute } \beta\ \beta_A) \rightsquigarrow \\ \qquad\qquad (\text{abstraction } (\tau_A, \{\ \}) \hookrightarrow (\tau'_A, \beta'_A))\end{array}}$$

(ABSTRACTION-I)

$$\frac{\varepsilon \vdash A\text{: } (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \textsf{abstraction } A\text{: abstraction } (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

Appendix C

Syntax of ACTRESS Specifications

grammar:

```
(1)  Symbols        =  Symbol ⟨ "," Symbol ⟩* .

(2)  Symbol         =  ⟨ ⟨ Syntactic-symbol | Semantic-symbol ⟩ "_"? ⟩ |
                       ⟨ Semantic-symbol "(" "_" ⟨ "," "_" ⟩* ")" ⟩ .

(3)  Constructor    =  ⟨ Syntactic-symbol Argument* ⟩ .

(4)  Variable       =  Syntactic-var | Semantic-var .

(5)  Argument       =  [[ ⟨ Syntactic-var ":" ⟩? Syntactic-symbol ]] .

(6)  Formula        =  [[ Term Relator Term ⟨ "(" Disjoiner ")" ⟩? ]] .

(7)  Relator        =  "=" | "=<" | ":" | ">=" | ":–" .

(8)  Disjoiner      =  "disjoint" | "individual" .

(9)  Clause         =  Formula | [[ Symbol "::" Functionality ]] .

(10) Functionality  =  [[ Terms "–>" Term ⟨ "(" Attribute ⟨ "," Attribute ⟩* ⟩? ]] .

(11) Attribute      =  [[ "total" ]] | [[ "partial" ]] | [[ "restricted" ]] | [[ "strict" ]] |
                       [[ "linear" ]] | [[ "associative" ]] | [[ "commutative" ]] |
                       [[ "idempotent" ]] | [[ "unit" "is" Term ]] .

(12) Basic          =  [[ "privately"? "introduces:" Symbols "." ]]
                       [[ ⟨ "includes:" | "needs:" ⟩ References "." ]]
                       [[ ⟨ Equation-label? Clause ⟩? "." ]] |
                       [[ "closed" "." ] | [[ "open" "." ]] |
                       [[ "closed" "except" References "." ]] .

(13) References     =  Reference ⟨ "," Reference ⟩* .

(14) Reference      =  Path | [[ Path "(" Translation ⟨ "," Translation ⟩* ")" ]] .
```

(15) Path = Title | [["/" Title]] | [[Path "/" Title]] |
[[Path "/" "(" Path ⟨ "," Path ⟩* ")"]] .

(16) Title = Title-word+ | "*" .

(17) Translation = [[Symbol ⟨ "for" Symbol ⟩$^{?}$]] .

(18) Module = [[Module-label Module-path Rule Specification]] |
[["grammar:" ⟨ Basic^{+} | Module^{+} ⟩]] .

(19) Module-path = Path ⟨ "(" "continued" ")" ⟩$^{?}$.

(20) Specification = [[Basic^{+}]] | [[Module^{+}]] | [[Basic^{+} Module^{+}]] .

(21) Terms = ⟨ Term ⟨ "," Term ⟩* ⟩ .

(22) Term = Prefix-term | [[Prefix-term Infix-symbol Prefix-term]] .

(23) Prefix-term = Postfix-term | [[Prefix-symbol Prefix-Term]] |
[["bind" Prefix-term "to" Prefix-term]] |
[["recursively" "bind" Prefix-term "to" Prefix-term]] |
[["store" Prefix-term "in" Prefix-term]] |
[["give" Prefix-term ⟨ "label" "#" Natural ⟩$^{?}$]] |
[["the" Prefix-term ⟨ "#" Natural ⟩$^{?}$]] |
[["the" Prefix-term "bound" "to" Prefix-term]] |
[["the" Prefix-term "stored" "in" Prefix-term]] |
[["if" Prefix-term "then" Prefix-term
"else" Prefix-term]] |
[[Semantic-symbol Prefix-term]] |
[[Semantic-symbol "(" Terms ")"]] .

(24) Postfix-term = Simple-term |
[[Postfix-term "[" ⟨ Outcomes | ⟨ "using" Incomes ⟩ |
Term ⟩ "]"]] .

(25) Simple-term = [["abstraction"]] | [["action"]] | [["commit"]] |
[["complete"]] | [["current" "bindings"]] |
[["current" "data"]] | [["current" "storage"]] |
[["diverge"]] | [["escape"]] | [["fail"]] | [["it"]] |
[["rebind"]] | [["regive"]] | [["unfold"]] | [["[]"]] |
[[Integer]] | [[Natural]] | [[Token]] |
[[Semantic-symbol]] | [[Syntactic-symbol]] |
[[Variable]] | [["(" Term ")"]] | [["[[" Constructor "]]"]] .

(26) Infix-symbol = "!" | "&" | "and" | "and" "then" |
"and" "then" "moreover" | "before" | "else" | "hence" |
"is" | "or" | "then" | "then" "moreover" | "thence" |
"trap" | "with" .

(27) Prefix-symbol = "an" | "of" | "yielder" | "allocate" | "check" | "choose" |
"deallocate" | "enact" | "furthermore" | "indivisibly" |
"reflect" | "reflection" | "reserve" | "unfolding" |
"unreserve" | "unstore" | "abstraction" | "closure" |

			"reflection" .
(28)	Outcomes	=	Outcome ⟨ "!" Outcome ⟩* .
(29)	Outcome	=	[["giving" Giving]] | [["giving" "(" Giving "," Giving ⟩$^+$ ")"]] | [["binding"]] | [["storing"]] | [["diverging"]] | [["failing"]] | [["completing"]] .
(30)	Giving	=	[["an" Prefix-Term ⟨ "label" "#" Natural ⟩$^?$]] .
(31)	Incomes	=	Income ⟨ "!" Income ⟩* .
(32)	Income	=	[["current" "bindings"]] | [["current" "storage"]] | [["the" Given]] | [["the" "(" Given ⟨ "," Given ⟩$^+$ ")"]] .
(33)	Given	=	[["the" Prefix-Term ⟨ "label" "#" Natural ⟩$^?$]] .
(34)	Syntactic-symbol	=	⟨ Upper ⟨ Letter | Digit | '–' ⟩$^+$ ⟩ .
(35)	Semantic-symbol	=	⟨ Lower ⟨ Letter | Digit | '–' ⟩$^+$ ⟩ .
(36)	Title-word	=	⟨ Letter | '–' ⟩$^+$.
(37)	Syntactic-var	=	⟨ ⟨ ⟨ "~" Upper Letter$^+$ "_" ⟩ | Upper ⟩ Digit* "'"* ⟩ .
(38)	Semantic-var	=	⟨ ⟨ ⟨ "~" Lower Letter$^+$ "_" ⟩ | Lower ⟩ Digit* "'"* ⟩ .
(39)	Natural	=	Digit$^+$.
(40)	Integer	=	⟨ ('+'|'–') Natural ⟩ .
(41)	Equation-label	=	⟨ "(" Digit$^+$ ⟨ "." Digit$^+$ ⟩* ")" ⟩ | ⟨ "(" "*" ")" ⟩ .
(42)	Module-label	=	⟨ ⟨ Upper | Digit$^+$ ⟩ ⟨ "." Digit$^+$ ⟩* ⟩ .
(43)	Rule	=	⟨ "_" "_"$^+$ ⟩ .

Appendix D

Proofs

D.1 Commutativity of meet

We want to prove that "*meet*" is commutative, i.e. that *meet* $S_1\ S_2 =$ *meet* $S_2\ S_1$.

Proof: We begin by constructing Table D.1 showing the corresponding result of the *meet* operation for each type of argument sort. If the entries in this table are symmetical about the leading diagonal, then the *meet* operation is commutative. From inspection, it is clear that the majority of cases are indeed symmetrical (given a simple renaming of the variables and an inductive hypothesis that *meet* is commutative). There are, however, four cases which are not obviously equivalent. These entries in Table D.1 have been highlighted, and we will consider them in more detail below.

Case 1: *meet* $(P_1 \mid S_1')$ nothing and *meet* nothing $(P_1 \mid S_1')$

$$
\begin{aligned}
\textit{meet}\ (P_1 \mid S_1')\ \text{nothing} &= (\textit{meet}\ P_1\ \text{nothing}) \mid (\textit{meet}\ S_1'\ \text{nothing}) \\
&= \text{nothing} \mid \text{nothing} \\
&= \text{nothing} \\
\textit{meet}\ \text{nothing}\ (P_1 \mid S_1') &= \text{nothing}
\end{aligned}
$$

Table D.1: Results of *meet* for different types of arguments

	nothing	datum	I_1	B_1	$C_1[S_1']$	$P_1 \mid S_1'$	S_{1a} & S_{1b}
nothing	nothing	nothing	nothing	nothing	nothing	(*meet* P_1 nothing) $\mid$ (*meet* S_1' nothing)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* S_1' nothing
datum	nothing	datum	I_1	B_1	$C_1[$*meet* datum $S_1']$	(*meet* P_1 datum) $\mid$ (*meet* S_1' datum)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* S_1' datum
I_2	nothing	I_2	if $I_1 = I_2$ then I_1 else nothing	if $I_2 \in B_1$ then I_2 else nothing	nothing	(*meet* $P_1\ I_2$) $\mid$ (*meet* $S_1'\ I_2$)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* $S_1'\ I_2$
B_2	nothing	B_2	if $I_1 \in B_2$ then I_1 else nothing	if $B_1 = B_2$ then B_1 else nothing	nothing	(*meet* $P_1\ B_2$) $\mid$ (*meet* $S_1'\ B_2$)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* $S_1'\ B_2$
$C_2[S_2']$	nothing	$C_2[$*meet* datum $S_2']$	nothing	nothing	if $C_1 = C_2$ then $C_1[$*meet* $S_1'\ S_2']$ else nothing	(*meet* $P_1\ C_2[S_2']$) $\mid$ (*meet* $S_1'\ C_2[S_2']$)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* $S_1'\ C_2[S_2']$
$P_2 \mid S_2'$	nothing	(*meet* datum P_2) $\mid$ (*meet* datum S_2')	(*meet* $I_1\ P_2$) $\mid$ (*meet* $I_1\ S_2'$)	(*meet* $B_1\ P_2$) $\mid$ (*meet* $B_1\ S_2'$)	(*meet* $C_1[S_1']\ P_2$) $\mid$ (*meet* $C_1[S_1']\ S_2'$)	(*meet* $P_1\ (P_2 \mid S_2')$) $\mid$ (*meet* $S_1'\ (P_2 \mid S_2')$)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* $S_1'\ (P_2 \mid S_2')$
S_{2a} & S_{2b}	nothing	*meet* $S_{2a}\ S_{2b}$	let $S_2' =$ *meet* $S_{2a}\ S_{2b}$ in *meet* $I_1\ S_2'$	let $S_2' =$ *meet* $S_{2a}\ S_{2b}$ in *meet* $B_1\ S_2'$	let $S_2' =$ *meet* $S_{2a}\ S_{2b}$ in *meet* $C_1[S_1']\ S_2'$	let $S_2' =$ *meet* $S_{2a}\ S_{2b}$ in (*meet* $S_1'\ S_2'$) $\mid$ (*meet* $P_2\ S_2'$)	let $S_1' =$ *meet* $S_{1a}\ S_{1b}$ in *meet* $S_1'\ (S_{2a}$ & $S_{2b})$

Case 2: $meet\ (S_{1a}\ \&\ S_{1b})$ nothing and $meet$ nothing $(S_{1a}\ \&\ S_{1b})$

$$\begin{aligned}
meet\ (S_{1a}\ \&\ S_{1b})\ \text{nothing} &= \text{let } S_1' = meet\ S_{1a}\ S_{1b} \text{ in } meet\ S_1'\ \text{nothing} \\
&= \text{let } S_1' = meet\ S_{1a}\ S_{1b} \text{ in nothing} \\
&= \text{nothing} \\
meet\ \text{nothing}\ (S_{1a}\ \&\ S_{1b}) &= \text{nothing}
\end{aligned}$$

Case 3: $meet\ (S_{1a}\ \&\ S_{1b})$ datum and $meet$ datum $(S_{1a}\ \&\ S_{1b})$

$$\begin{aligned}
meet\ (S_{1a}\ \&\ S_{1b})\ \text{datum} &= \text{let } S_1' = meet\ S_{1a}\ S_{1b} \text{ in } meet\ S_1'\ \text{datum} \\
&= S_1' \\
meet\ \text{datum}\ (S_{1a}\ \&\ S_{1b}) &= meet\ S_{1a}\ S_{1b} \\
&= S_1'
\end{aligned}$$

Case 4: $meet\ (P_1 \mid S_1')\ (S_{2a}\ \&\ S_{2b})$ and $meet\ (S_{2a}\ \&\ S_{2b})\ (P_1 \mid S_1')$

$$\begin{aligned}
meet\ (P_1 \mid S_1')\ (S_{2a}\ \&\ S_{2b}) &= \text{let } S_2' = meet\ S_{2a}\ S_{2b} \\
&\quad \text{in } (meet\ P_1\ S_2') \mid (meet\ S_1'\ S_2') \\
meet\ (S_{2a}\ \&\ S_{2b})\ (P_1 \mid S_1') &= \text{let } S_2' = meet\ S_{2a}\ S_{2b} \text{ in } meet\ S_2'\ (P_1 \mid S_1') \\
&= \text{let } S_2' = meet\ S_{2a}\ S_{2b} \\
&\quad \text{in } (meet\ S_2'\ P_1) \mid (meet\ S_2'\ S_1') \\
&= \text{let } S_2' = meet\ S_{2a}\ S_{2b} \\
&\quad \text{in } (meet\ P_1\ S_2') \mid (meet\ S_1'\ S_2')
\end{aligned}$$

Therefore, we have shown that Table D.1 is indeed symmetrical, and so *meet* is commutative.

Note that from Table D.1, it also easy to see that the result of *meet* does not include any occurences of the "&" operator. Each entry in the table is either trivially of the correct format, or involves the further application of *meet* to the sub-components of the arguments. Therefore the *meet* algorithm traverses the entire structure of both arguments, and eliminates all occurrences of "&" in the result.

D.2 Normalisation of normalise

We want to prove that "*normalise S*" results in a sort of the form $S_1 \mid \dots \mid S_n$, where $n > 0$, and none of the S_i contain occurrences of "&", i.e. that it produces a sort in normal form.

Proof: The proof is constructed using structural induction over the syntax of sorts.

Case 1: *normalise* nothing = nothing

Trivially in normal form.

Case 2: *normalise* datum = datum

Trivially in normal form.

Case 3: *normalise* I = I

Trivially in normal form.

Case 4: *normalise* B = B

Trivially in normal form.

Case 5: *normalise* $C[S]$ = let S' = *normalise* S in $C[S']$

By the inductive hypothesis, S' is in normal form, and therefore, so is $C[S']$.

Case 6: *normalise* $(S_1 \ \& \ S_2)$ = let S' = *meet* $S_1 \ S_2$ in *normalise* S'

From the properties of *meet*, S' will not contain any occurrences of "&", and by the inductive hypothesis *normalise* S' will be in normal form.

Case 7: *normalise* $(P_1 \mid S_2)$ = let P_1' = *normalise* P_1

S_2' = *normalise* S_2

in

prune $(P_1' \mid S_2')$

By the inductive hypothesis, both P_1' and S_2' are in normal form, and therefore, so is *prune* $(P_1' \mid S_2')$.