1990

Type inference for action semantics

Susan Johanna Even
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd
Part of the Computer Sciences Commons

Recommended Citation
INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Type inference for action semantics

Even, Susan Johanna, Ph.D.

Iowa State University, 1990
Type inference for action semantics

by

Susan Johanna Even

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Approved:  
Signature was redacted for privacy.

In Charge of Major Work
Signature was redacted for privacy.

For the Major Department
Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1990
# TABLE OF CONTENTS

## CHAPTER 1. INTRODUCTION

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>1</td>
</tr>
<tr>
<td>Semantics of action semantics</td>
<td>2</td>
</tr>
<tr>
<td>Type inference for action semantics</td>
<td>3</td>
</tr>
<tr>
<td>Background</td>
<td>5</td>
</tr>
<tr>
<td>Denotational semantics</td>
<td>5</td>
</tr>
<tr>
<td>Drawbacks to the lambda calculus metalanguage</td>
<td>6</td>
</tr>
<tr>
<td>Action semantics</td>
<td>10</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>19</td>
</tr>
<tr>
<td>Type inference</td>
<td>21</td>
</tr>
</tbody>
</table>

## CHAPTER 2. THE SEMANTICS OF ACTIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facets and Subtyping</td>
<td>23</td>
</tr>
<tr>
<td>Interpretation of Facets</td>
<td>26</td>
</tr>
<tr>
<td>Formalization as functors</td>
<td>29</td>
</tr>
<tr>
<td>Kinds</td>
<td>30</td>
</tr>
<tr>
<td>Interpretation of Actions</td>
<td>31</td>
</tr>
<tr>
<td>Primitives</td>
<td>32</td>
</tr>
<tr>
<td>Typing functions</td>
<td>35</td>
</tr>
<tr>
<td>Chapter 3. Type Inference</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ML-style, Unification-based Type Inference</td>
<td>45</td>
</tr>
<tr>
<td>Type Schemes</td>
<td>48</td>
</tr>
<tr>
<td>Facets</td>
<td>48</td>
</tr>
<tr>
<td>Primitive actions</td>
<td>50</td>
</tr>
<tr>
<td>Soundness and completeness</td>
<td>51</td>
</tr>
<tr>
<td>Type Inference for Action Combinators</td>
<td>54</td>
</tr>
<tr>
<td>Sequential composition</td>
<td>54</td>
</tr>
<tr>
<td>Parallel composition</td>
<td>59</td>
</tr>
<tr>
<td>Concatenation composition</td>
<td>65</td>
</tr>
<tr>
<td>Simplifying properties</td>
<td>68</td>
</tr>
<tr>
<td>Conditional composition</td>
<td>70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4. The Satisfaction of Constraints</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Inference With Subtypes</td>
<td>74</td>
</tr>
<tr>
<td>Fuh and Mishra’s Satisfaction Algorithm</td>
<td>75</td>
</tr>
<tr>
<td>Consistency of atomic constraints</td>
<td>76</td>
</tr>
<tr>
<td>Extension to Record and Storage Type Constraints</td>
<td>78</td>
</tr>
<tr>
<td>Distinguished constraint forms</td>
<td>78</td>
</tr>
<tr>
<td>Decomposition</td>
<td>79</td>
</tr>
<tr>
<td>Simplification of a constraint</td>
<td>81</td>
</tr>
</tbody>
</table>
Decomposition of storage ........................................ 84
Decomposition of records ....................................... 85
Weakness of constraints .......................................... 89
Partial correctness of decomposition ........................ 90
Satisfaction of distinguished constraints .................... 94
Rowlessness and invisibility ..................................... 96

CHAPTER 5. APPLICATION AND IMPLEMENTATION ........ 97
Specialized Action Semantics ................................... 97
Facets .................................................................. 97
Primitive actions .................................................... 98
Command and expression abstracts .......................... 101
An example language definition ............................... 105
The IN combinator .................................................. 107
An example of type inference .................................. 109
Implementation of the Type Inference Algorithms ........ 120

CHAPTER 6. CONCLUSION ........................................ 126
Related Work ....................................................... 126
Milner-style type inference ..................................... 126
Subtyping ............................................................ 128
Records ................................................................ 129
Ongoing and Future Research .................................. 136

BIBLIOGRAPHY ..................................................... 139
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Abstract Syntax</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Semantic Domains</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Valuation Functions</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>Action Semantics</td>
<td>15</td>
</tr>
<tr>
<td>1.5</td>
<td>Mosses-style Action Semantics</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>Interpretation of Facets</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Kinds</td>
<td>31</td>
</tr>
<tr>
<td>2.3</td>
<td>Typing Functions for Primitives</td>
<td>33</td>
</tr>
<tr>
<td>2.4</td>
<td>Meaning Functions for Primitives</td>
<td>34</td>
</tr>
<tr>
<td>2.5</td>
<td>Flow of Arguments for Action Combinators</td>
<td>37</td>
</tr>
<tr>
<td>2.6</td>
<td>Kindings for Action Combinators</td>
<td>38</td>
</tr>
<tr>
<td>2.7</td>
<td>Typing Functions for Action Combinators</td>
<td>39</td>
</tr>
<tr>
<td>2.8</td>
<td>Meaning Functions for Action Combinators</td>
<td>39</td>
</tr>
<tr>
<td>2.9</td>
<td>Flow of Arguments for Derived Composition</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td>Syntax of Type Schemes</td>
<td>49</td>
</tr>
<tr>
<td>3.2</td>
<td>Typing Schemes for Primitive Actions</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Unification of Record Type Schemes</td>
<td>56</td>
</tr>
</tbody>
</table>
vi

Figure 3.4. Unification of Storage Type Schemes .................................. 57
Figure 3.5. Type Inference for Merge on the Declarative Facet .......... 61
Figure 3.6. Type Inference for Concatenation ................................. 66

Figure 4.1. Fuh and Mishra's algorithm CONSISTENT ....................... 77
Figure 4.2. Algorithms decompose and decompose-constraint .......... 80
Figure 4.3. Algorithm simplify .............................................. 81
Figure 4.4. Decomposition of Storage ........................................ 85
Figure 4.5. Decomposition of Records ........................................ 86

Figure 5.1. Specialized Facets .................................................. 99
Figure 5.2. Interpretation of the Functional Facet ......................... 100
Figure 5.3. Specialized Schemes for Primitives ............................. 100
Figure 5.4. Semantics of Abstracts .......................................... 101
Figure 5.5. Schemes for Abstracts ............................................. 104
Figure 5.6. An Example Language Definition: Syntax .................... 105
Figure 5.7. An Example Language Definition: Action Semantics ...... 106
Figure 5.8. The Example Program ............................................. 109
Figure 5.9. ML Implementation of Facets ................................... 121
Figure 5.10. ML Implementation of Schemes ................................. 122
Figure 5.11. ML Implementation of find i and bind i ..................... 123
Figure 5.12. ML Implementation of Sequential Composition ............ 125

Figure 6.1. Advances in Type Schemes ....................................... 136
CHAPTER 1. INTRODUCTION

Overview

Action semantics is a metalanguage for describing the denotational semantics of programming languages [21, 22, 23, 24, 25, 27]. Action semantics was introduced by Mosses as an alternative to the lambda calculus notation traditionally used as the metalanguage. In action semantics, the meaning of a language construct is denoted by an "action." Primitive actions exist for all the fundamental operations of programming languages: value passing and arithmetic, binding creation and lookup, storage allocation and updating, and so on. Actions are composed to denote more complex computations.

Actions operate on "facets." Mosses distinguishes facets for each of the different kinds of information that arise during computation. These include a functional facet for temporary values ("transient" information), a declarative facet for identifier-value bindings ("scoped" information), and an imperative facet for storage cells ("stable" information), among others. In action semantics, facets are envisioned as "flowing through" semantics equations; emphasis is on the information itself rather than the data structures responsible for its maintenance, such as environments and stores. An important feature of action semantics is that it hides semantic domain details from the reader.
The focus of our research has been the study of actions as polymorphic combinators that operate on collections of types. Our work includes a category-sorted algebra-based model for action semantics; a unification-based type inference algorithm for action expressions similar to that used for ML, extended with subtypes and records; proofs of its soundness and completeness with respect to the model; and an algorithm for simplifying inheritance subtyping constraints on records to constraints on non-record primitives.

We now preview the remaining chapters with a summary of our results.

Semantics of action semantics

Our model for action semantics is based on Reynolds's category-sorted algebra, extended to "many" sorts to accommodate the different facets in action semantics [8, 32, 33]. Actions are formalized as polymorphic combinators that operate on collections of types. Types in the functional facet include primitive, function and product types; types in the declarative and imperative facets are records. The type names in each facet are partially ordered to reflect subtyping relationships such as \( \text{int} \leq \text{real} \). Our subtype orderings are based on Cardelli [3] and Cardelli and Wegner's [5] work. For each combination of facets \( K \), an interpretation functor \( A_K \) maps the collection of type names in \( K \) to semantic domains, and the subtype ordering on type names to coercion functions between domains. As in Reynolds's work, our coercion functions are not limited to embeddings.

An action \( a \) is interpreted as a family of mappings, indexed by the type names in its argument facets. Each action has a typing function \( T_a \) that maps each type in its argument facets to a result type.
its typing function: if \( a \) requires arguments in facets \( K_1 \) to produce results in facets \( K_2 \), then for each type \( k \in K_1 \), \( a(k) \) has type \( k \rightarrow T_a(k) \) and its interpretation is a mapping from the domain \( A_{K_1}[[k]] \) to the domain \( A_{K_2}[T_a(k)] \).

Chapter 2 presents the details of our semantic model.

**Type inference for action semantics**

We have developed type schemes for actions that express their polymorphism, and a unification-based type inference algorithm with subtypes and records for expressions in action semantics. The algorithm determines the type of an action expression from the context in which it appears. An action’s typing function is complex and cumbersome to work, whereas its typing scheme is simple and expresses in schematic form the non-ns behavior of the action. Having a type inference system for action semantics greatly enhances its usability.

Our type inference algorithm is based on the work of a number of researchers: Milner’s polymorphic type inference for ML [18]; Mitchell [20] and Fuh and Mishra’s [10, 11] work on type inference with subtypes; and the work on type inference with records by Wand [41, 42], Stanisifer [38], Jategaonkar and Mitchell [15], and Remy [31]. Our work extends and improves upon each of these systems, primarily because our results are based on a semantic model. We have the following results:

- We prove both the soundness and completeness of our type inference algorithm with respect to our model for action semantics.

- We avoid the large constraint sets encountered by Mitchell [20] and Fuh and Mishra [10, 11] in their system. Coercions are not needed in our model: if an action \( a \) is defined on values of type \( t_2 \), then for any subtype \( t \leq t_2 \), there is
a morphism $a(t) : t \rightarrow T_a(t)$. We don’t need to coerce a value of type $t$ to a value of type $t_2$ before applying the action. Therefore, we do not introduce a subtyping constraint.

- Our system provides record concatenation and union operations, without the need for complex constraints on record types encountered by Wand [42]. The typing schemes assigned action expressions satisfy simplifying properties that “restrict away” the problem cases.

- We give an algorithm for decomposing constraints on records related by inheritance subtyping into constraints on non-record primitive types and prove its partial correctness. The algorithm extends the algorithms given by Fuh and Mishra [10] for structural subtyping with subtyping on primitive types. Stansifer omits such an algorithm from [38].

- Our semantic model and our type inference algorithm accommodate several forms of polymorphism: ML-style, parametric polymorphism, inclusion polymorphism in the form of subtyping on primitives and inheritance subtyping on records [3, 5], as well as a form of ad-hoc polymorphism [5].

- We have implemented all of our type inference algorithms in ML.

Chapters 3 and 4 present our type inference algorithms. Chapter 5 presents an application of the type inference algorithm and discusses its implementation in ML. Chapter 6 compares our type inference system with the systems mentioned above.
Background

One of the goals of action semantics is to make the semantics notation itself more understandable. In the remainder of this chapter, we informally describe the action semantics metalanguage to give the reader an intuitive idea of what actions do in terms of information flow through an action semantics expression. We also present background material on denotational semantics, polymorphism, and type inference.

Denotational semantics

Denotational semantics [17, 37, 39] assigns mathematical functions to program code in a syntax-directed manner. A denotational semantics definition consists of three parts: the abstract syntax specification of the language, the semantic domains, and the valuation functions.

The abstract syntax is specified by listing syntax domains plus the BNF rules for constructing elements of each non-primitive syntax domain. An abstract syntax for a small imperative language is given in Figure 1.1.

The meanings of programs and the values that they operate on are elements of semantic domains. A semantic domain may be either primitive or compound. Compound domains, built from primitive domains, include the product domain, the function domain, and the disjoint union domain. Semantic domains for the language in Figure 1.1 plus the operations on the domains are given in Figure 1.2.

To each syntax domain, we assign a valuation function. Valuation functions map elements in syntax domains to elements in semantic domains. Valuation functions for the example language are given in Figure 1.3. Valuation functions are listed as a set of equations, one corresponding to each choice in the BNF rule. The portion on the
right-hand side of an equation is called the “denotation” of the language construct. Denotations are higher-order functions, written in a semantic metalanguage. The metalanguage is usually the lambda calculus [1, 39].

**Drawbacks to the lambda calculus metalanguage**

Mosses criticizes the use of the lambda calculus notation as a metalanguage [21]. He observes that denotational semantics is generally presented “as if the meaning of each construct was just the application of higher-order functions to each other,” and that function application is used to represent a number of different concepts:

**Look-ups in environments and stores** With the traditional representation of environments and stores as functions:

\[
\begin{align*}
\text{Environment} &= \text{Id} \rightarrow \text{Denotable-value} \\
\text{Store} &= \text{Loc} \rightarrow \text{Storable-value}
\end{align*}
\]
\[ n \in \text{Nat} \]
\[ b \in \text{Bool} \]
\[ i \in \text{Id} \quad \text{(Identifiers)} \]
\[ \ell \in \text{Loc} \quad \text{(Locations)} \]
\[ \text{firstlocn: Loc} \]
\[ \text{nextlocn: Loc} \rightarrow \text{Loc} \]

\[ d \in \text{Denotable-value} = \text{Nat} + \text{Loc} \]
\[ e \in \text{Environment} = \text{Id} \rightarrow \text{Denotable-value} \]
\[ \text{newenv: Environment} \]
\[ \text{newenv} = \lambda i . \text{"undefined"} \]
\[ \text{find: Id} \rightarrow \text{Environment} \rightarrow \text{Denotable-value} \]
\[ \text{find} = \lambda i . \lambda e . e i \]
\[ \text{bind: Id} \rightarrow \text{Denotable-value} \rightarrow \text{Environment} \rightarrow \text{Environment} \]
\[ \text{bind} = \lambda i . \lambda d . \lambda e . [i \mapsto d] e \]

\[ m \in \text{Memory} = \text{Loc} \rightarrow \text{Nat} \]
\[ s \in \text{Store} = \text{Loc} \times \text{Memory} \]
\[ \text{newstore: Store} \]
\[ \text{newstore} = (\text{firstlocn}, \lambda \ell . \text{zero}) \]
\[ \text{contents: Loc} \rightarrow \text{Store} \rightarrow \text{Nat} \]
\[ \text{contents} = \lambda \ell . \lambda (\ell', m) . m \ell \]
\[ \text{update: Loc} \rightarrow \text{Nat} \rightarrow \text{Store} \rightarrow \text{Store} \]
\[ \text{update} = \lambda \ell . \lambda n . \lambda (\ell', m) . (\ell', [\ell \mapsto n] m) \]
\[ \text{allocate: Store} \rightarrow \text{Loc} \times \text{Store} \]
\[ \text{allocate} = \lambda (\ell, m) . (\ell, (\text{nextlocn} \ell, m)) \]

Figure 1.2. Semantic Domains (Unbound identifiers are ignored for simplicity.)
\( N \): Numeral \( \rightarrow \) Nat (omitted)  
\( B \): Boolean \( \rightarrow \) Environment \( \rightarrow \) Store \( \rightarrow \) Bool (omitted)  

\( \mathcal{E} \): Expression \( \rightarrow \) Environment \( \rightarrow \) Store \( \rightarrow \) Nat  
\[ \mathcal{E}[I] = \lambda e \cdot \lambda s \cdot \text{cases find } I e \text{ of} \]  
\[ \quad \text{isNat}(n) \rightarrow n \big| \]  
\[ \quad \text{isLoc}(l) \rightarrow \text{contents } l s \]  
\[ \mathcal{E}[N] = \lambda e \cdot \lambda s \cdot N[N] \]  
\[ \mathcal{E}[E_1 + E_2] = \lambda e \cdot \lambda s \cdot (\mathcal{E}[E_1]e s) \text{ plus } (\mathcal{E}[E_2]e s) \]  

\( \mathcal{D} \): Declaration \( \rightarrow \) Environment \( \times \) Store \( \rightarrow \) Environment \( \times \) Store  
\[ \mathcal{D}[D_1 ; D_2] = \mathcal{D}[D_2] \circ \mathcal{D}[D_1] \]  
\[ \mathcal{D}[\text{const } I = E] = \lambda(e,s) \cdot \text{let } n = \mathcal{E}[E]e s \text{ in} \]  
\[ \quad \text{(bind } I \text{ in } \text{Nat}(n) e, s) \]  
\[ \mathcal{D}[\text{var } I] = \lambda(e,s) \cdot \text{let } (\ell, s') = \text{allocate } s \text{ in} \]  
\[ \quad \text{(bind } I \text{ inLoc}(\ell) e, s') \]  

\( \mathcal{C} \): Command \( \rightarrow \) Environment \( \rightarrow \) Store \( \rightarrow \) Store  
\[ \mathcal{C}[K] = \mathcal{K}[K] \]  
\[ \mathcal{C}[I := E] = \lambda e \cdot \lambda s \cdot \text{cases find } I e \text{ of} \]  
\[ \quad \text{isNat}(n) \rightarrow \text{"error"} \big| \]  
\[ \quad \text{isLoc}(\ell) \rightarrow \text{let } n = (\mathcal{E}[E]e s) \text{ in} \]  
\[ \quad \text{update } \ell e n s \]  
\[ \mathcal{C}[C_1 ; C_2] = \lambda e \cdot (\mathcal{C}[C_2]e) \circ (\mathcal{C}[C_1]e) \]  
\[ \mathcal{C}[\text{if } B \text{ then } C_1 \text{ else } C_2] = \]  
\[ \quad \lambda e \cdot \lambda s \cdot (B[B]e s) \rightarrow (C[C_1]e s) \big| (C[C_2]e s) \]  

\( \mathcal{K} \): Block \( \rightarrow \) Environment \( \rightarrow \) Store \( \rightarrow \) Store  
\[ \mathcal{K}[\text{begin } D \text{ in } C \text{ end}] = \]  
\[ \quad \lambda e \cdot \lambda s \cdot \text{let } (e', s') = \mathcal{D}[D](e, s) \text{ in} \]  
\[ \quad \mathcal{C}[C]e' s' \]  

Figure 1.3. Valuation Functions (Error cases are omitted for simplicity.)
lookup in both domains is represented by function application. For example,

\[ E[I] = \lambda e . \lambda s . \text{cases } (e I) \text{ of} \]

\[ \text{isLoc}(\ell) \rightarrow (s \ell) | \ldots \]

**Sequencing and order of evaluation** Sequential execution is represented by function composition:

\[ C : \text{Command} \rightarrow \text{Store} \rightarrow \text{Store} \]

\[ C[C_1 ; C_2] = \lambda s . C[C_2](C[C_1]s) \]

Possible concurrent execution is observed when different constructs are applied to the same bound instance of a variable:

\[ E : \text{Expression} \rightarrow \text{Environment} \rightarrow \text{Nat} \]

\[ E[E_1 + E_2] = \lambda e . (E[E_1]e) \text{ plus } (E[E_2] e) \]

Both sub-constructs are applied to the same environment, used in a read-only mode.

**Scope rules and binding strategies** Static and dynamic scoping, and parameter binding strategies are both expressed by whether or not a denotation is applied to an environment and a store, and when. For example, a command denotation is applied to an environment at the point of definition for static scoping:

\[ D : \text{Declaration} \rightarrow \text{Environment} \rightarrow \text{Environment} \]

\[ D[\text{proc } I = C] = \lambda e . \text{bind } I \text{ inProc}(C[C]e) e \]
For dynamic scoping, C's denotation is not applied to an environment at
definition—it is applied to an environment at the time of call.

The reader must recognize these patterns of application to infer operational properties
of the semantics from the metalanguage.

Action semantics

Mosses has introduced action semantics [21, 22, 23, 24, 25, 27] as an alternative
to lambda calculus-style denotational semantics. Denotations of program phrases are
now expressed as "actions" rather than as functions in lambda notation.

Actions initially appeared as auxiliary operations. Auxiliary operations are often
defined on semantic domains to abbreviate complex pieces of lambda notation. This
makes the semantic description more concise. Auxiliary operations are also referred
to as combinators [1]. A typical combinator introduced is for sequencing:

_ then _ : CDen × CDen → CDen
c1 then c2 = λs . λe . c2 e (c1 e s)

where CDen = Environment → Store → Store is a semantic domain for command
denotations. This combinator may now be used in command valuation:

\[ C[C_1 ; C_2] = C[C_1] \text{ then } C[C_2] \]

An advantage of using combinators is that semantic definitions no longer contain lists
of arguments.

Mosses uses a set of auxiliary operations in [23] to specify the semantics of a
small tutorial language. He then gives both direct and continuation-style definitions
to the operations without modifying the semantic equations. The semantic equations are not committed to any model due to the abstractness of the operations.

A goal of action semantics is to have a general set of combinators which can be used to describe the semantics of a wide range of languages and constructs. Further, the interpretation given to the actions may be changed without modifying the semantic metalanguage.

Facets Mosses describes actions in terms of the "facets" they operate on. Facets are the result of distinguishing the different semantic domains used in denotational semantics and isolating the portions of the definitions which operate on each. This results in (at least) three facets which operate on sets of values, identifier bindings, and storage cells. The facets are referred to as the functional, declarative, and imperative facets, respectively. Mosses describes the facets in terms of what type of information flows through each facet, without reference to the data structure responsible for its maintenance. This avoids commitment to models of information storage.

Actions Actions may be either single-faceted or multi-faceted. Single-faceted actions are concerned with just one of the facets and do not use or produce information in the other facets. Multi-faceted actions may use information from and produce information in more than one facet.

Primitives There are both single-faceted and multi-faceted primitives. Single-faceted primitives pass information along a facet. They may remove a binding from the set of bindings, select values from the values stream, or apply coercions and
operations on primitive semantic domains such as plus to values.

Multi-faceted primitives are called bridging actions. Bridging actions interface more than one facet. Actions which bridge the functional and binding facets allow us to establish new bindings and and retrieve values bound to identifiers:

\[ \text{bind I: Action [consuming values] [making bindings]} \]
\[ \text{find I: Action [using bindings] [producing values]} \]

These actions are analogous to their namesakes in Figure 1.2. The “types” of the actions are deliberately left vague.

Primitives which bridge the functional and imperative facets allow us to create new storage cells and update and examine their contents:

\[ \text{alloc: Action [inspecting storage]} \]
\[ [\text{producing values} [\text{modifying storage}]} \]
\[ \text{update: Action [consuming values] [inspecting storage]} \]
\[ [\text{modifying storage}]} \]
\[ \text{contents: Action [consuming values] [inspecting storage]} \]
\[ [\text{producing values}]} \]

update and contents each require arguments from both the functional and the imperative facets.

The imperative and declarative facets are not directly bridged—they are bridged only indirectly through the functional facet when actions are combined.

**Action combinators** Just as there are single-faceted and multi-faceted primitives, there are single-faceted and multi-faceted action combinators. Tied to action combinators is the idea of facet flow, which we discuss first.
The combination of two actions results in a composite action. There are two ways the facets may "flow through" a composite action: either horizontally (with the combinator ':', ')

"arguments" \rightarrow a_1 \rightarrow ; \rightarrow a_2 \rightarrow "results"

or vertically (with the combinator 'v', 'v'):

\[
\begin{array}{c}
\text{"arguments"} \\
\uparrow \\
a_1 \quad * \\
\downarrow \\
\text{"results"}
\end{array}
\]

The words "horizontal" and "vertical" reflect the way the arrows are drawn to show the flow of the facets through the component actions.

Horizontal flow corresponds to sequencing and may be thought of as ordinary function composition. Vertical flow corresponds to concurrent valuation. Information passed to the composite action is duplicated and passed to each component action. Each action operates independently to produce results which are then "merged." In the values facet, values are concatenated. In the bindings facet, the sets of bindings produced by the component actions are unioned, with the restriction that the sets of bindings must be disjoint.

Vertical flow in the imperative facet introduces interfering parallelism. Since the stores produced must be interleaved, we are required to model concurrency with something such as resumption semantics [29, 37]. For this reason, we disallow the merging of storage. However, storage may flow vertically when both component actions do not produce results in the storage facet.
In general, information from more than one facet may flow through a composite action and the action may combine both horizontal and vertical flow. Combinators like then can be used to direct facet flows through composite actions. For example, \( a_1 \) then \( a_2 \) directs bindings information vertically into \( a_1 \) and \( a_2 \), and directs storage information horizontally through \( a_1 \) and \( a_2 \).

We use a general format for combinators: the symbol "§" is suffixed with designators for those facets which flow horizontally. Thus then is really §\(_I\), where \( I \) indicates the imperative facet. A well-formedness constraint on \( a_1 \) §\(_k\) \( a_2 \) requires that \( a_1 \) produce information in the facets indicated by \( k \).

Figure 1.4 gives an action semantics for the language in Figure 1.1. Looking at the semantic equation for the assignment statement,

\[
C[I := E] = (\text{find } I \ast E[E]) \ S_F \text{ update}
\]

we see that the identifier look-up find \( I \) on the incoming bindings, and the valuation of the expression \( E[E] \) are performed in parallel, with their results tupled. The storage cell, value pair on the functional facet is then directed sequentially into the update action, whose storage argument is passed vertically from the arguments to the composite action. The result of the composite action is the modified storage argument.

Notice that most of the typing information for the actions in Figure 1.4 is omitted. For example, in the above equation, we deduce that valuation of find \( I \) must result in a storage cell rather than a number. The types actions possess are cumbersome (as we will see in the next chapter), but our type inference algorithm allows an action to be used with as few explicit typing annotations as possible.
\[ E : \text{Expression} \rightarrow \text{Action} \begin{cases} \text{using bindings} & \text{[inspecting storage]} \\
\text{producing values} & \end{cases} \]

\[ E[I] = \text{find}_I / (\text{find}_I \uparrow_E \text{contents}) \]
\[ E[N] = \text{put}_\text{nat}(N[N]) \]
\[ E[E_1 + E_2] = (E[E_1] \times E[E_2]) \uparrow_E \text{add}_\text{nat} \]

\[ D : \text{Declaration} \rightarrow \text{Action} \begin{cases} \text{using bindings} & \text{[inspecting storage]} \\
\text{making bindings} & \text{[modifying storage]} & \end{cases} \]

\[ D[D_1 ; D_2] = D[D_1] \downarrow_{DF} D[D_2] \]
\[ D[\text{const } I = E] = (E[E] \uparrow_E \text{rebind } I) \star \text{skip} \]
\[ D[\text{var } I] = \text{alloc}_\text{nat} \uparrow_E \text{rebind } I \]

\[ C : \text{Command} \rightarrow \text{Action} \begin{cases} \text{using bindings} & \text{[inspecting storage]} \\
\text{modifying storage} & \end{cases} \]

\[ C[K] = K[K] \]
\[ C[I := E] = (\text{find}_I \star E[E]) \uparrow_E \text{update} \]
\[ C[C_1 ; C_2] = C[C_1] \downarrow_E C[C_2] \]
\[ C[\text{if } B \text{ then } C_1 \text{ else } C_2] = B[B] \uparrow_E \text{choose}(C[C_1], C[C_2]) \]

\[ K : \text{Block} \rightarrow \text{Action} \begin{cases} \text{using bindings} & \text{[inspecting storage]} \\
\text{modifying storage} & \end{cases} \]

\[ K[\text{begin } D \text{ in } C \text{ end}] = D[D] \downarrow_{DF} C[C] \]

Figure 1.4. Action Semantics
Evaluate: Expression $\rightarrow$ Action [using bindings] [inspecting storage] [producing values]

$$\text{Evaluate}\left[ E_1 + E_2 \right] =$$

$$(\text{Evaluate}[E_1] \text{ and } \text{Evaluate}[E_2]) \text{ then } \text{Operate}[+]$$

Establish: Declaration $\rightarrow$ Action [using bindings] [inspecting storage] [making bindings] [modifying storage]

$$\text{Establish}\left[ D_1 ; D_2 \right] = \text{Establish}[D_1] \text{ before } \text{Establish}[D_2]$$

$$\text{Establish}[\text{var I}] =$$

allocate a_number_variable then bind I to the_variable

Execute: Command $\rightarrow$ Action [using bindings] [inspecting storage] [modifying storage]

$$\text{Execute}[C_1 ; C_2] = \text{Execute}[C_1] \text{ and then } \text{Execute}[C_2]$$

Figure 1.5. Mosses-style Action Semantics

**Features of action semantics**

1. **Action semantics is easy to read.** Semantic domain details are hidden from the user. The more recent versions of the action semantics notation have a natural language flavor: denotations resemble “instructions” in COBOL programs [25, 27]. (See Figure 1.5.) An action semantics expression thus conveys some intuitive operational ideas, but the technical details are obscured. We prefer to work with an earlier, less verbose version of the notation whose mathematical properties are much easier to state [22]. However, we use the general combinator $\&$, defined in terms of sequential (;) and parallel (*) composition, to define other forms of composition. (This combinator supercedes ‘;’, ‘‘;’, and ‘!’.)
2. Action semantics conveys operational properties. It is easier to note the operational implications of a language definition when it is expressed in action semantics. Action semantics solves the defects Mosses noted about the use of function application in the lambda calculus notation. There are actions for doing look-ups in environments and stores, and there are actions that explicitly freeze identifier bindings and thus clarify forms of scoping. Sequencing and order of evaluation are both specified by facet flow: actions can be "glued together" as if connecting up wires through which different kinds of information flow, thus coloring function application with the kind of information operated upon.

3. Action semantics is modular and reusable. The same semantic expressions can be used without modification in defining different languages. For example, if two languages have arithmetic expressions, the respective action semantics of them should look the same. The modularity of action semantics has been demonstrated by Mosses and others with sample language definitions. Mark [16] shows that it is possible to reuse parts of the semantic descriptions for ML, with only minor editing, to describe the language AMBER [6]. Actions liberate semantic equations from the semantic domains underneath, and also provide the semantic equations with independence from the language context in which they are used.

4. Action semantics clarifies language concepts. Some language concepts might be more easily understood from the study of semantic definitions expressed in action semantics. We believe action semantics could be used to detect important language properties or concepts such as whether a language is dynamically or statically scoped, sequential imperative [36], statically typed, or block-structured
Actions have more detailed typing annotations than those usually found in denotational semantics definitions. For example, in addition to type annotations related to the values that flow through semantics equations, there are annotations for sets of bindings that should be present in an environment and locations that should be present in a store. Our ongoing and future research includes the study of how types such as these indicate language structure.

The modularity of action semantics  What makes action semantics modular? We believe a key idea is that an action is a “polymorphic” entity [18]. The word “polymorphic” means “having many types.” Actions are polymorphic with respect to the facets they require arguments in. For example, \texttt{find} \texttt{I} requires information from the bindings facet to produce information in the values facet. However, \texttt{find} \texttt{I} can operate when applied to other facets in addition to the bindings facet. It just ignores or throws away information in the extra facets. Actions are also polymorphic with respect to individual facets: \texttt{find} \texttt{I} can operate when applied to any set of bindings that contains a binding for \texttt{I}.

Implicit coercions also play an important role in the modularity of action semantics. When there is a natural injection from one domain into another, such as a coercion from natural numbers into integers, this is represented by an implicit coercion operator. There are also implicit coercions among the traditional sum domains of denotable, expressible and storable values. For example, suppose each summand of \texttt{Denotable-value} is also a summand of \texttt{Expressible-value}. Then for the semantic equation:

\[ \mathcal{E} [\texttt{I}] = \texttt{find} \texttt{I} \]
the denotable value produced by \textit{find} I is implicitly coerced into an expressible value. Further, if \textit{Nat} is a summand of both \textit{Expressible-value} and \textit{Denotable-value}, then a value of type \textit{Nat} may be coerced into either of these domains.

\textbf{Polymorphism}

The focus of our research has been to investigate actions and action combinators as polymorphic entities. Cardelli and Wegner distinguish several different kinds of polymorphism [5]. Functions which are "generic" exhibit \textit{parametric polymorphism}. This form of polymorphism is obtained when a function works uniformly on a range of types that share some common structure. For example, the function \textit{fst} = \lambda(a, b) . a may be applied to any value which is a pair, regardless of what types its components have. Parametric polymorphism allows the same function to be used uniformly in different type contexts, without coercions. This is the form of polymorphism found in ML [18, 19].

Operator overloading is a form of \textit{ad-hoc polymorphism}: polymorphism that is "added on later." With overloading, the same identifier is used to stand for different functions and the context of its use determines which function it denotes. For example, the identifier \textit{click} could be used to represent the successor operation \textit{succ} on numbers and the negation operation \textit{not} on boolean values. With overloading, neither the operation nor the values it operates on need be polymorphic, and the types of the arguments may not even be related.

Coercions are another form of \textit{ad-hoc polymorphism}. A coercion is a semantic operation that converts an argument to the type expected by a function. For example, a function on real numbers may be applied to an integer by first applying a coercion
(either implicit or explicit) to the integer argument. Coercions are sometimes used to allow a monomorphic (single-typed) operation to be used as if it were polymorphic. A value could also be made polymorphic: for example, a coercion could be defined to convert strings into booleans.

Cardelli and Wegner point out that the distinction between the two forms of ad-hoc polymorphism, coercion and overloading, often blurs. For example, given the expressions $2+3$, $2.0+3$, $2+3.0$, and $2.0+3.0$, we could explain the polymorphism of the operator $+$ as four overloaded meanings with types $\text{int} \times \text{int} \rightarrow \text{int}$, $\text{real} \times \text{int} \rightarrow \text{real}$, $\text{int} \times \text{real} \rightarrow \text{real}$, and $\text{real} \times \text{real} \rightarrow \text{real}$, or as two overloaded meanings with types $\text{int} \times \text{int} \rightarrow \text{int}$ and $\text{real} \times \text{real} \rightarrow \text{real}$, where if one of the arguments is an integer and the other a real, the integer is coerced to a real.

Inclusion polymorphism is used to model subtypes and inheritance. A value whose type $\tau_1$ is a subtype of another type $\tau_2$ may be used in any context where a value of type $\tau_2$ is expected. Subtyping allows a value to have many types. We may also think of a value of type $\tau_1$ as “inheriting” all of those operations that are defined on types $\tau_2$ which are supertypes of $\tau_1$. For example, if the type brie is a subtype of the type cheese, then all of the operations on values of type cheese can be applied to a value of type brie.

An important issue with polymorphism is the question of whether it is the values which are polymorphic or the operations on the values which are polymorphic, or both. To answer this question, we must consider the semantics of the types and their operations. In action semantics, the actions are the polymorphic entities and the values they operate upon are monomorphic. Actions exhibit each of the different forms of polymorphism described above. We study the semantics of actions and their
polymorphism in Chapter 2.

Type inference

A polymorphic function may have infinitely many legal typings. For example, the function \( \text{apply}(f, x) = f(x) \) is legally typed whenever \( f \) has a functional type \( t_1 \rightarrow t_2 \), for some types \( t_1 \) and \( t_2 \), and \( x \) has type \( t_1 \). The result of the application \( f(x) \) will then have type \( t_2 \). Thus we can infer for any arbitrary types \( t_1 \) and \( t_2 \) that \( \text{apply} \) has type \( ((t_1 \rightarrow t_2) \times t_1) \rightarrow t_2 \).

One way to represent an infinite set of types is by using a type scheme. A type scheme is a type expression that may contain type variables. The type scheme \( ((\alpha \rightarrow \beta) \times \alpha) \rightarrow \beta \), where \( \alpha \) and \( \beta \) are type variables, abbreviates the infinite set of types of \( \text{apply} \). Legal typings such as \( ((\text{int} \rightarrow \text{bool}) \times \text{int}) \rightarrow \text{bool} \) are obtained by substituting other type expressions for the type variables.

Type inference is the process of inferring the types of program phrases by their use. Having a type inference system for a language allows us to omit many typing annotations. The above example illustrates parametric polymorphic type inference, which originated with the programming language ML. In ML, each expression has a most general type (also called a principal type scheme). This means that for each program phrase, there exists a single "best" type expression that abbreviates all its legal typings.

Mitchell [20] and Fuh and Mishra [10] extend ML-style type inference to include subtypes. In the presence of subtypes, we cannot represent the set of all typings of a function by a type expression alone. For example, suppose a language has the types \text{int}, \text{real}, and \text{bool}, and that \text{int} is a subtype of \text{real}, written \text{int} \subseteq \text{real}. Suppose
we define an operation $\textit{succ} = \lambda n \cdot n + 1$. Although $\textit{succ}$ has types $\text{int} \to \text{int}$ and $\text{real} \to \text{real}$, we cannot express the type of $\textit{succ}$ using the scheme $\theta \to \theta$, because substituting the type $\text{bool}$ for the type variable $\theta$ would produce an erroneous type $\text{bool} \to \text{bool}$.

To remedy this, polymorphic types are represented by a pair consisting of a type expression and a set of coercions. The coercions are subtyping constraints on the variables in the type expression. Any substitution that makes the constraints hold true can be applied to the type expression to yield an instance of the type. For example, we can use the scheme $\theta \to \theta \text{ if } \{ \theta \leq \text{real} \}$ to abbreviate the legal typings of $\textit{succ}$. Instantiating $\theta$ to $\text{int}$ produces the legal typing $\text{int} \to \text{int}$ because the constraint $\text{int} \leq \text{real}$ is true.

We wish to point out that another type $\textit{succ}$ could possess is $\text{int} \to \text{real}$, since $\text{int}$ is a subtype of $\text{real}$ [10]. This type is not an instance of $\theta \to \theta \text{ if } \{ \theta \leq \text{real} \}$. (Fuh and Mishra would assign $\textit{succ}$ a scheme of the form $\theta_1 \to \theta_2 \text{ if } \{ \theta_1 \leq \text{real}, \theta_1 \leq \theta_2 \}$.) We choose to disallow this typing because it is not as precise: an implicit coercion has been applied to the result and typing information has been lost.
CHAPTER 2. THE SEMANTICS OF ACTIONS

In this chapter, we present the semantics for our version of the action semantics metalanguage. We first formalize the concept of "facet" described in Chapter 1. These are the semantic domains on which actions operate. We then discuss the interpretation of actions as polymorphic combinators over the facets. Details of the model may be found in [8].

Facets and Subtyping

A facet is a collection of types that are partially ordered to reflect subtyping relationships [32, 33]. The functional facet is the collection of all types that can be used as temporary values in a computation. Mosses calls this "transient information." We use the following types in the functional facet:

\[ t \in \text{Functional-facet-type} \]
\[ p \in \text{Primitive-type} \]
\[ t ::= p \mid t_1 \rightarrow t_2 \mid t_1 \times t_2 \mid (\ell : t)\text{var} \]
\[ p ::= \text{nat} \mid \text{int} \mid \text{real} \mid \text{bool} \]

\((\ell : t)\text{var}\) is the type of storage variables, where \(\ell\) is a location. The types defined by the BNF rules are called "proper types." The collection of proper types in each facet
will be unioned with an additional "improper" type called ns ("nonsense") which
stands for an undefined type.

The subtype ordering on the functional facet is the smallest reflexive, transitive
ordering such that:

\[ t \leq \text{ns}, \text{ for all } t \]
\[ \text{nat} \leq \text{int} \]
\[ \text{int} \leq \text{real} \]

\[ t_1 \times t_2 \leq t'_1 \times t'_2 \text{ if and only if } t_1 \leq t'_1 \text{ and } t_2 \leq t'_2 \]

\[ t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2 \text{ if and only if } t'_1 \leq t_1 \text{ and } t_2 \leq t'_2 \]

The ordering on products and functions is that found in [5]. The subtyping on
function types allows a function such as / with type \text{real} - \text{int} to be used in a
context that expects a function with type \text{int} - \text{real}, since / can be applied to
arguments of type \text{int}, and its results of type \text{int} also have type \text{real}.

The declarative facet contains types of identifier-value bindings. Mosses calls this
"scoped information." The proper types in the declarative facet are records [3, 5]:

\[ d \in \text{Declarative-facet-type} \]
\[ d := \{i : t_i\}_{i \in I} | \{i : t_i\}_{i \in I}^{exactly} \]

where \( I \) is a finite set of identifiers. We call \( I \) the index set of the record. It
is customary to read the type \( \{i : t_i\}_{i \in I} \) as the type of records that have at least
bindings of type \( t_i \) for the identifiers \( i \) in \( I \) (and may have additional bindings) and
the type \( \{i : t_i\}_{i \in I}^{exactly} \) as the type of records that have bindings for exactly the
identifiers in \( I \) and no others.
The subtype ordering on the declarative facet is the smallest reflexive, transitive ordering such that:

\[
d \leq \text{ns}, \quad \text{for all } d
\]

\[
\{i: t_i\}_{i \in I} \leq \{j: t'_j\}_{j \in J} \quad \text{if and only if} \quad J \subseteq I \quad \text{and} \quad \forall j \in J, \ t_j \leq t'_j
\]

\[
\{i: t_i\}_{i \in \text{exactly}} \leq \{i: t_i\}_{i \in I}
\]

\[
\{i: t_i\}_{i \in \text{exactly}} \leq \{i: t'_i\}_{i \in \text{exactly}} \quad \text{if and only if} \quad \forall i \in I, \ t_i \leq t'_i
\]

The \{i: t_i\}_{i \in I}\text{-type records use inheritance subtyping [3], whereas the \{i: t_i\}_{i \in \text{exactly}}\text{-type records do not. With inheritance subtyping, a function defined on records with at least a binding for B to a value with type real (e.g., a function with type } {B: \text{real}} \rightarrow \text{real} \text{) can be applied to an argument record with other bindings besides B (e.g., an argument with type } {B: \text{int}, C: \text{bool}} \text{). We have use for both forms of record type in action semantics.}

The imperative facet contains types of storage structure or shape. Mosses calls this “stable information.” The proper types in the imperative facet are sets of storage cells:

\[
s \in \text{Imperative-facet-type}
\]

\[
s ::= \langle \ell: t_\ell \rangle_{\ell \in L}
\]

where \( L \) is a finite set of locations. The subtype ordering on the imperative facet is the smallest reflexive, transitive ordering such that:

\[
s \leq \text{ns}, \quad \text{for all } s
\]

\[
\langle \ell: t_\ell \rangle_{\ell \in L} \leq \langle k: t_k \rangle_{k \in K} \quad \text{if and only if} \quad K \subseteq L
\]
One storage shape is a supertype of another when all of its storage cells are contained in the subtype. The ordering is the same as that for non-exactly records in the declarative facet, however, subtyping is not used on storage cell contents.

Note that the types in the declarative and imperative facets are dependent on the types in the functional facet, and the primitive types are dependent on the language being defined in action semantics. The declarative and imperative facets will show up in the functional facet in Chapter 5 when we use action semantics to denote procedure and macro definitions. We have not yet studied the use of subsets of the functional facet such as "denotable" and "storable" values.

**Interpretation of Facets**

Figure 2.1 gives the interpretation of the proper types and the subtype orderings for each facet. Each type is interpreted as a predomain. The primitive types in the functional facet are interpreted as the sets of natural numbers, integers, rational numbers, and boolean values. The type constructors 'x' and '→' are interpreted as the product and function domain constructions on the interpretation of their component types. The variable type (ℓ:t)var is interpreted as a pair of access and update functions, with domains:

\[
\text{access}_{t,\ell} : A_F[(\ell : t)] \rightarrow A_F[t]
\]

\[
\text{update}_{t,\ell} : A_F[t] \rightarrow A_F[(\ell : t)] \rightarrow A_F[(\ell : t)]
\]

The functions operate on a single storage cell in the imperative facet with location ℓ and type t, since the subtyping on the imperative facet will allow the functions to operate on store shapes that are larger than a single-celled store. More will be said on this when we explain the actions alloc, contents, and update.
THE FUNCTIONAL FACET

\[ A_F[\text{nat}] = \mathbb{N} \]
\[ A_F[\text{int}] = \mathbb{Z} \]
\[ A_F[\text{real}] = \mathbb{Q} \]
\[ A_F[\text{bool}] = \mathbb{B} \]
\[ A_F[t_1 \rightarrow t_2] = A_F[t_1] \rightarrow A_F[t_2] \]
\[ A_F[t_1 \times t_2] = A_F[t_1] \times A_F[t_2] \]
\[ A_F[(\ell : t)\text{var}] = (A_F[(\ell : t)] \rightarrow A_F[t]) \times (A_F[t] \rightarrow A_F[(\ell : t)]) \]

\[ A_F[\text{nat} \leq \text{int}] = \lambda n. n \]
\[ A_F[\text{int} \leq \text{real}] = \lambda n. n \]
\[ A_F[t_1 \rightarrow t_2 \leq t'_2 \rightarrow t'_1] = \lambda f. A_F[t_2 \leq t'_2] \circ f \circ A_F[t'_1 \leq t_1] \]
\[ A_F[t_1 \times t_2 \leq t'_1 \times t'_2] = A_F[t_1 \leq t'_1] \times A_F[t_2 \leq t'_2] \]

Figure 2.1. Interpretation of Facets

The subtype ordering for each facet is interpreted into coercion functions on the interpreted types [32, 33]. The coercions on the primitives are embeddings for the relations \( \text{nat} \leq \text{int} \) and \( \text{int} \leq \text{real} \). The coercion on a product type is the product of the coercion functions on the component types, where the product of functions \( f \) and \( g \) is defined as \( f \times g (a, b) = (f a, g b) \). The coercion of a function \( f \) to a supertype "inserts" coercions on both the argument and the result of \( f \).

In the declarative facet, we interpret the record types \( \{i: t_i\}_{i \in I} \) and \( \{i: t_i\}_{i \in I} \) exactly the same way: as indexed products of the interpretations of each of the types of the identifiers in the record's index set. The subtype ordering is the crucial feature of records. The coercion on non-exactly records projects out the components of the subtype which are also components of the supertype, applying the appropriate coercion to each. In contrast, the coercions on exactly records do not alter the structure of a
THE DECLARATIVE FACET

\[ AD[\{i : t_i\}_{i \in I}] = \prod_{i \in I} A_F[t_i] \]
\[ AD[\{i : t_i\}_{i \in I} \text{exactly}] = \prod_{i \in I} A_F[t_i] \]

\[ AD[\{i : t_i\}_{i \in I} \leq \{j : t_j'\}_{j \in J}] = \lambda r. \prod_{j \in J} A_F[t_j \leq t_j'](r \downarrow j) \]
\[ AD[\{i : t_i\}_{i \in I} \text{exactly} \leq \{i : t_i\}_{i \in I}] = \lambda r. \prod_{i \in I} A_F[t_i \leq t_i'](r \downarrow i) \]

THE IMPERATIVE FACET

\[ AI[\langle \ell : t_\ell \rangle_{\ell \in L}] = \prod_{\ell \in L} A_F[t_\ell] \]
\[ AI[\langle \ell : t_\ell \rangle_{\ell \in L} \leq \langle k : t_k \rangle_{k \in K}] = (\text{drop, lift}) \]

where \( \text{drop}: AI[\langle \ell : t_\ell \rangle_{\ell \in L}] \rightarrow AI[\langle k : t_k \rangle_{k \in K}] \)
\[ \text{drop} = \lambda p. \prod_{k \in K} (p \downarrow k) \]

and \( \text{lift}: (AI[\langle k : t_k \rangle_{k \in K}] \rightarrow AI[\langle k : t_k \rangle_{k \in K}]) \rightarrow AI[\langle \ell : t_\ell \rangle_{\ell \in L}] \rightarrow AI[\langle \ell : t_\ell \rangle_{\ell \in L}] \)
\[ \text{lift} = \lambda c. \lambda p. \lambda \ell. \text{let } p' = c (\prod_{k \in K} (p \downarrow k)) \]
\[ \text{in } \prod_{\ell \in L} (\text{if } \ell \in K \text{ then } (p' \downarrow \ell) \text{ else } (p \downarrow \ell)) \]

Figure 2.1 (Continued)
record, they merely apply coercions to its components.

In the imperative facet, each storage shape is interpreted as the indexed product of the interpretations of each location's type. The subtype ordering is interpreted as a pair of functions: a function called \textit{drop} which truncates a storage shape by dropping off cells whose locations are not defined in the supertype; and a function called \textit{lift} which transforms a function defined over values of the supertype into a function that works on the subtype. \textit{drop} is defined like the coercion on non-exact record types. \textit{lift} coerces an argument store (of the subtype) to the type expected by the function (defined on the supertype) by dropping off the unneeded storage cells. It applies the function and then produces a result store (of the subtype) by tupling the unused cells of the argument with the cells modified by the function.

We have seen that variables are interpreted as a pair of access-update functions on a single storage cell. The \textit{drop} and \textit{lift} functions allow us to apply a variable's access and update functions to any shape store that contains the variable's cell. More will be said on this later. The functions will also come in useful when we define procedures in Chapter 5.

For each facet, the improper type \textquoteleft ns\textquoteright is interpreted as the singleton set \{\textit{()},\}, and a coercion into \textit{ns} is the constant function that maps all arguments to \textit{()}. 

\textbf{Formalization as functors}

The interpretation of the types and subtyping relation is formalized as a functor from the partial ordering of type names, treated as a category, into the category \textit{Pdom} of predomains [13]. A category is a collection of "objects" and "morphisms." Graphically, the morphisms are drawn as arrows connecting the objects. A functor is
a special kind of function defined on categories that maps each object in its domain category to an object in its codomain category, and each morphism in its domain category to a morphism in its codomain category. The interpretation functors in Figure 2.1 map each type name in the partial ordering of types to a predomain, and each subtyping relation between two type names to a continuous function between their corresponding predomains. We denote the interpretation functors for each facet by $A_F$, $A_D$, and $A_I$.

Functors naturally preserve the reflexive and transitive properties of the subtype ordering on type names in their interpretation: reflexivity corresponds to the identity mapping and transitivity corresponds to function composition. For example, by showing that $A_F$ is indeed a functor, we are assured that the coercions behave as expected: if $\text{nat} \leq \text{int}$ and $\text{int} \leq \text{real}$, we can coerce a value of type $\text{nat}$ to an $\text{int}$ and then to a $\text{real}$, or coerce the value directly to a $\text{real}$ and obtain the same result in either case. We have seen that the coercions need not be mere embeddings; they can be quite complex. However, the coercions are never explicitly or implicitly inserted in action expressions—they remain hidden from a user of the notation. More will be said on this later.

Kinds

To accommodate multi-faceted actions, additional structure is added to the interpretation of the facets. We use the ordering given in Figure 2.2 on combinations of facet names called kinds. The ordering states that information in unneeded facets can be "forgotten." For example, a functional, declarative pair can be coerced into just a functional facet argument or just a declarative facet argument.
The kinds $F$, $D$, and $I$ are interpreted as the collections of types we saw earlier. Each combination of facets, $FD$, $FI$, $DI$, and $FDI$, is interpreted as the product of the collections of types in the individual facets. $I$ is the "empty facet"; it contains the types $\text{ok}$ and $\text{ns}$, and has the ordering $\text{ok} \leq \text{ns}$.

The interpretation functor for each combination of facets is interpreted as the product of the interpretation functors on the individual facets. For example, $A_{FD} = A_F \times A_D$. For clarity, we will use $\times$ to stand for the notion of product on facets. The interpretation functor for the empty facet is defined as: $A_t[\text{ok}] = A_t[\text{ns}] = \{()\}$, and $A_t[\text{ok} \leq \text{ns}] = \lambda().()$.

**Interpretation of Actions**

Actions are mappings whose domains and codomains are facets. We call the domain of an action its source, and its codomain its target. For example, we write $\text{copy}: F \rightarrow F$ to state that the source and target of action $\text{copy}$ are the functional facet ("$F$"). The declarative facet will be represented by a "$D$", and the imperative facet will be represented by an "$I$". We call the specification of an action's source
and target facets its *kinding*.

Since a facet contains many types, actions are polymorphic mappings on the cpos that the types denote. Each action has a *typing function* that describes its behavior on argument types. For example, *copy* is the identity mapping on the functional facet, so its typing function is $T_{\text{copy}} = (\lambda t : F . t)$, which states that *copy* maps an argument of type $t$ to an answer of type $t$. *copy* exhibits parametric polymorphism.

The meaning of *copy* is the family of identity functions \( \{\lambda v : t . v\}_{t \in F} \), which we represent by \( A_{\text{copy}} = \lambda t : F . \lambda v : t . v \). We call \( A_{\text{copy}} \) the action's *meaning function*.

**Primitives**

The kindings, typing functions, and meaning functions for the set of primitive and bridging actions we use are given in Figures 2.3 and 2.4. *put* accepts input along the empty facet and emits value $v$ with type $t_0$ as its result. Note that because values in our model of action semantics are monomorphic, action *put* must be indexed with the type of its argument. *succ* increments arguments from \( \text{nat} \), \( \text{int} \) and \( \text{real} \) and is undefined on nonnumbers. For example, $T_{\text{succ}}(\text{int}) = \text{int}$, since $\text{int} \leq \text{real}$, but $T_{\text{succ}}(\text{bool}) = \text{ns}$, since $\text{bool} \not\leq \text{real}$.

*pass* is the identity mapping for the declarative facet. *bind i* builds a declarative record that holds exactly the one binding of identifier $i$ to the argument received by the action. *find i* retrieves from its declarative record argument the binding to $i$. $T_{\text{find i}}$ reveals the inclusion polymorphism of *find i*: its result type is non-ns for any record type that has at least a field for $i$. For example, $T_{\text{find A}}(\{A: \text{int}, B: \text{bool}\}) = \text{if } \{A: \text{int}, B: \text{bool}\} \leq \{A: t\} \text{ then } t \text{ else } \text{ns}$. The free identifier $t$ becomes bound to $\text{int}$ when the constraint is found true. Notice that *find i* exhibits parametric


<table>
<thead>
<tr>
<th>ACTION</th>
<th>KINDING</th>
<th>TYPING FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(_{(v:t_0)})</td>
<td>1 \rightarrow F</td>
<td>(T_{\text{put}_{(v:t_0)}}) = (t_0)</td>
</tr>
<tr>
<td>copy</td>
<td>F \rightarrow F</td>
<td>(T_{\text{copy}}) = (t)</td>
</tr>
<tr>
<td>not</td>
<td>F \rightarrow F</td>
<td>(T_{\text{not}}) = ( \text{if } t = \text{bool then bool else ns} )</td>
</tr>
<tr>
<td>succ</td>
<td>F \rightarrow F</td>
<td>(T_{\text{succ}}) = ( \text{if } t \leq \text{real then } t \text{ else ns} )</td>
</tr>
<tr>
<td>pass</td>
<td>D \rightarrow D</td>
<td>(T_{\text{pass}}) = (d)</td>
</tr>
<tr>
<td>bind (i)</td>
<td>F \rightarrow D</td>
<td>(T_{\text{bind}}_i) = ({i : t})(\text{exactly})</td>
</tr>
<tr>
<td>find (i)</td>
<td>D \rightarrow F</td>
<td>(T_{\text{find}}_i) = ( \text{if } d \leq {i : t} \text{ then } t \text{ else ns} )</td>
</tr>
<tr>
<td>skip</td>
<td>I \rightarrow I</td>
<td>(T_{\text{skip}}) = (s)</td>
</tr>
<tr>
<td>contents</td>
<td>FI \rightarrow F</td>
<td>(T_{\text{contents}}_t'_s) =</td>
</tr>
<tr>
<td>update</td>
<td>FI \rightarrow I</td>
<td>(T_{\text{update}}_t''_s) =</td>
</tr>
<tr>
<td>alloc(_{t_0})</td>
<td>I \rightarrow FI</td>
<td>(T_{\text{alloc}}_s) = (\langle \ell : t_0\rangle \text{var} \times \langle \ell : t_0\rangle s), where (\ell = \text{newlocn})</td>
</tr>
</tbody>
</table>

Figure 2.3. Typing Functions for Primitives
MEANING FUNCTION

\[ A_{\text{put}(v)}(t) = \lambda v . v \]
\[ A_{\text{copy}}(t) = \lambda v : t . v \]
\[ A_{\text{not}}(t) = \lambda v : t . \neg v \]
\[ A_{\text{succ}}(t) = \lambda v : t . v + 1 \]
\[ A_{\text{pass}}(d) = \lambda r : d . r \]
\[ A_{\text{bind}}(t) = \lambda v : t . \{ i = v \} \]
\[ A_{\text{find}}(d) = \lambda r : d . r \downarrow i \]
\[ A_{\text{skip}}(s) = \lambda p : s . p \]
\[ A_{\text{contents}}((\ell : t)\text{var}, s) = \lambda (f, p) . (f \downarrow 1) \left( (A_{F}[s \leq (\ell : t)] \downarrow 1) \right) p \]
\[ A_{\text{update}}((\ell : t)\text{var} \times t', s) = \lambda ((f, v), p) . (A_{F}[s \leq (\ell : t)] \downarrow 2) ((f \downarrow 2) v) p \]
\[ A_{\text{alloc}}(s) = \lambda p : s . (f, (\ell = \text{init}_{t_0}) p) \]
where \( f = (\lambda (\ell = v) . v, \lambda v . \lambda (\ell = v') . (\ell = v)) \), \( \ell = \text{newlocn}() \), and \( \text{init}_{t_0} \) is some fixed value in \( A_{F}[t_0] \).

Figure 2.4. Meaning Functions for Primitives (Note: for all actions \( a : K_1 \rightarrow K_2 \), for all types \( k \in K_1 \), if \( T_a(k) = \text{ns} \), then \( a(k) = \lambda v . () . \))

polymorphism on \( i \)'s binding.

\text{skip} is the identity mapping on the imperative facet. \text{contents} uses the variable on the functional facet to look inside its imperative facet argument. The first component of the variable, its access function, is applied to the variable's storage cell after the other cells have been dropped from the storage argument. For example, \( T_{\text{contents}}((\ell_2 : \text{int})\text{var}, (\ell_2 : \text{int}, \ell_3 : \text{real})) = \text{int} \), since \( (\ell_2 : \text{int}, \ell_3 : \text{real}) \leq (\ell_2 : \text{int}) \).

\text{contents}((\ell_2 : \text{int})\text{var}, (\ell_2 : \text{int}, \ell_3 : \text{real})) \left( f, p \right) \) applies the \text{drop} component of the coercion \( A_{F}[ (\ell_2 : \text{int}, \ell_3 : \text{real}) \leq (\ell_2 : \text{int}) ] \) to the storage argument \( p \) to isolate the variable's storage cell, and then applies the variable's access operation \( (f \downarrow 1) \) to it.

\text{update} uses the variable and the new storable value on the functional facet to
update its imperative facet argument. The update function of the variable is applied to the new value and then lifted by the \textit{lift} coercion from a function on a single storage cell to a function that operates on the shape of the incoming storage argument.

The actions \textit{contents} and \textit{update} exhibit inclusion polymorphism: an implicit coercion on the incoming storage argument is used before the monomorphic access and update functions belonging to the variable on the functional facet are applied. \textit{update} also applies a coercion to the new storable value. The use of the subset ordering on storage shapes allows us to recover the polymorphism of the two actions.

**Typing functions**

The typing functions specify the argument types that produce non-\texttt{ns} results, the structure required of an action's argument (i.e., \textit{update} requires its argument on the functional facet to be a pair in which the first component is a variable), and the coercions to be applied for the different argument types.

An important property of the typing functions is that they are monotone with respect to the subtype ordering: for all actions \texttt{a}, for all types \texttt{t1} and \texttt{t2} such that \texttt{t1} \leq \texttt{t2}, \texttt{T_a(t1)} \leq \texttt{T_a(t2)}. A consequence of this is that if \texttt{T_a(t1)} = \texttt{ns}, then \texttt{T_a(t2)} = \texttt{ns} as well. Thus, an action that is defined on type \texttt{t2}, i.e., produces non-\texttt{ns} results for arguments of type \texttt{t2}, must be defined on all subtypes of \texttt{t2}. For example, an action that is defined on reals must be defined on all subtypes of \texttt{real}. All of our actions are \texttt{ns-strict}: for all actions \texttt{a}, \texttt{T_a(ns)} = \texttt{ns} and \texttt{a(ns)} = \texttt{\lambda().()}. 
Meaning functions

The meaning of an action \( a: K_1 \rightarrow K_2 \) is formalized as a natural transformation from the functor \( A_{K_1} \) to \( A_{K_2} \circ T_a \), where \( A_{K_1} \) and \( A_{K_2} \) are the interpretation functors for \( a \)'s source and target. A natural transformation is an indexed family of mappings: \( \{ f_k : k \rightarrow T_a(k) \}_{k \in K_1} \). Look again at the meaning function \( A_{\text{copy}} = \lambda t : F . \lambda v : t . v \).

The notation "\( \lambda t : F \)” tells us \( \text{copy} \) is a family of mappings indexed by the types in the functional facet. The morphism \( \text{copy}(t) = \lambda v : t . v \) is a mapping with type \( t \rightarrow T_{\text{copy}}(t) \). \( \text{copy}(t) \) maps values in \( A_F[t] \) to values in \( A_F[T_{\text{copy}}(t)] \). The notation "\( \lambda v : t \)” indicates \( v \) has type \( t \).

An important property of an action's meaning function is that it respects the action's typing function. For types \( k \) and \( k' \) in \( K_1 \) such that \( k \leq k' \), the morphisms \( a(k) \) and \( a(k') \) commute with the coercion functions \( A_{K_1}[k \leq k'] \) and \( A_{K_2}[T_a(k) \leq T_a(k')] \) in the following way:

\[
\begin{array}{ccc}
A_{K_1}[k] & \xrightarrow{a(k)} & A_{K_2}[T_a(k)] \\
\downarrow A_{K_1}[k \leq k'] & & \downarrow A_{K_2}[T_a(k) \leq T_a(k')] \\
A_{K_1}[k'] & \xrightarrow{a(k')} & A_{K_2}[T_a(k')] \\
\end{array}
\]

The diagram shows us that it does not matter whether the coercion is performed on the argument to the action or on its result—the result is the same in either case.

Action composition

We study four forms of action composition: \( a_1 ; a_2 \), sequential composition; \( a_1 \ast a_2 \), parallel composition; \( a_1 \circ a_2 \), a restricted concatenation composition; and \( a_1 / a_2 \), conditional composition. Figure 2.5 shows the flow of arguments directed by
Figure 2.5. Flow of Arguments for Action Combinators

Each combinator. Sequential and parallel composition were introduced in Chapter 1.

The composition \( a \) represents concatenation composition on the declarative facet: the declarative record argument to the action is given to \( a \), and \( a \)'s output, which is another declarative record, is concatenated to the original record, overriding bindings in the original argument. For example, \( (\text{find A} ; \text{bind B}) \) accepts a record which is used to make a binding of B to A's value. The binding to B is concatenated to the original bindings. The form of concatenation composition given here is a restricted version of the binary composition \( a _1 \odot a _2 \), where the outputs of the two actions are concatenated. The unary form is actually \( \text{pass} \odot a \), which explains why we treat \( \odot a \) as a "composition". In the future, we plan to study a general form of the combinator for concatenation on all three facets.

The combinator \( / \) represents conditional composition. Arguments to the composite action are given to either \( a _1 \) or \( a _2 \), depending on which of the two is capable of mapping the arguments to a non-ns result. For example, \( \text{not} / \text{succ} \) maps a boolean argument to its negation, a numeric argument to its successor, and all other
<table>
<thead>
<tr>
<th>ACTION</th>
<th>KINDING</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a_1; a_2))</td>
<td>(K_1 \rightarrow K'_2)</td>
</tr>
<tr>
<td>((a_1 \ast a_2))</td>
<td>(K_1 \rightarrow K_2 \cap K'_2)</td>
</tr>
<tr>
<td>((a; a))</td>
<td>(K_1 \rightarrow D)</td>
</tr>
<tr>
<td>((a_1/a_2))</td>
<td>(K_1 \rightarrow K_2)</td>
</tr>
</tbody>
</table>

Figure 2.6. Kindings for Action Combinators

arguments to \(\text{ns}\). Conditional composition builds actions that exhibit ad-hoc polymorphism.

Figure 2.6 lists the kindings for each of the combinators. Their typing and meaning functions are defined in Figures 2.7 and 2.8.

Sequential composition composes the typing functions of the two actions. This has the effect of narrowing the collection of types on which the actions are defined, because the typing functions are \(\text{ns}\)-preserving: input types \(k\) for which \(T_{a_1}(k) = k' \neq \text{ns}\), but \(T_{a_2}(k') = \text{ns}\) are undefined for \(T_{a_1; a_2}\). In the definition of \(A_{a_1; a_2}\) we see that action \(a_2\)'s type parameter is the result of \(a_1\)'s typing function on its input. This shows how the morphisms of \(a_1\) and \(a_2\) are forced to respect the coercions: the only morphisms selected from \(a_2\) for composition with the morphisms of \(a_1\) are those which act upon values with legal result types from \(a_1\).

Parallel composition uses a meta-action called \(\text{merge}\). \(T_{\text{merge}}\) is a family of typing functions \(T_{\text{merge}}(K_2, K'_2)\), indexed by kinds \(K_2\) and \(K'_2\). Merging on the functional facet is pairing. Merging on the declarative facet is record union, but records with possible disagreements of field values are not unioned. The typing function \(T_{\text{merge}}(D, D)\) limits record union so that only records with exactly types can be unioned. This pre-
**Typing Function**

\[ T_{a_1, a_2} = T_{a_2} \circ T_{a_1} \]

\[ T_{a_1, a_2}(k) = T_{\text{merge}(K_2, K'_2)}(T_{a_1}(k), T_{a_2}(k)) \]

where \( T_{\text{merge}(F, F)}(t_1, t_2) = t_1 \times t_2 \),

and \( T_{\text{merge}(D, D)}(d_1, d_2) = \)

\[
\begin{align*}
&\text{if } (d_1 = \{i : t_i\}_{i \in \text{exactly}} \text{ and } d_2 = \{j : t_j\}_{j \in \text{exactly}} \text{ and } I \cap J = \emptyset) \\
&\text{then } \{i : t_i\} \oplus \{j : t_j\} \text{ exactly else } \text{ns}
\end{align*}
\]

\[ T_{a_1 / a_2}(k) = T_{\text{concat}(\text{declarative-facet-in}(k), T_{a_1}(k))} \]

where \( T_{\text{concat}(d_1, d_2) = \text{if } (d_2 = \{i : t_i\}_{i \in \text{exactly}} \text{ then } \{i : t_i\} \oplus (d_1 \setminus I) \text{ else } \text{ns}} \)

\[ T_{a_1 / a_2} = \text{if domain}(T_{a_1}) \cap \text{domain}(T_{a_2}) = \emptyset \]

\[ \text{then } (\lambda k : K_1 . \text{ if } T_{a_1}(k) \neq \text{ns then } a_1(k) \text{ else } a_2(k)) \]

\[ \text{else } T_{\text{fail}(K_2)} \circ T_{a_1} \]

**Meaning Function**

\[ A_{a_1, a_2}(k) = a_2 \left(T_{a_1}(k)\right) \circ a_1(k) \]

\[ A_{a_1, a_2}(k) = \lambda v : k . A_{\text{merge}(K_2, K'_2)}(T_{a_1}(k), T_{a_2}(k)) \left(a_1 k v, a_2 k v\right) \]

where \( A_{\text{merge}(F, F)}(t_1, t_2) = \lambda (v_1 : t_1, v_2 : t_2) \cdot (v_1, v_2), \)

and \( A_{\text{merge}(D, D)}(d_1, d_2) = \lambda (r_1 : d_1, r_2 : d_2) \cdot r_1 \cup r_2 \)

\[ A_{a_1 / a_2} = \lambda v : k . A_{\text{concat}(\text{record-component-of}(v), a k v)} \]

\[ A_{a_1 / a_2} = \text{if domain}(T_{a_1}) \cap \text{domain}(T_{a_2}) = \emptyset \]

\[ \text{then } (\lambda k : K_1 . \text{ if } T_{a_1}(k) \neq \text{ns then } a_1(k) \text{ else } a_2(k)) \]

\[ \text{else } (\lambda k : K_1 . A_{\text{fail}(K_2)}(T_{a_1}(k)) \circ a_1(k)) \]

**Figure 2.7.** Typing Functions for Action Combinators (Note: if either of the arguments to \( T_{\text{merge}(F, F)} \) or \( T_{\text{merge}(D, D)} \) is \( \text{ns} \), the result is \( \text{ns} \).)

**Figure 2.8.** Meaning Functions for Action Combinators
vents unseen clashes of "hidden" fields. Merging of imperative facet values has not been studied. A merge of different facets is defined as the component-wise merge of common facets and tupling of the rest. For example, \( T_{\text{merge}}(FD,DI)((t_1, d_1), (d_2, s)) = t \times T_{\text{merge}}(D,D)(d_1, d_2) \times s \).

Concatenation composition uses the typing function \( T_{\text{concat}} \) to limit concatenation so that the type of bindings that override existing bindings is an exactly type. This prevents hidden overriding.

Conditional composition requires that the domains of the two action's typing functions be disjoint, otherwise a \( \text{ns} \) value must be produced. For action \( a: K_1 \rightarrow K_2 \), \( \text{domain}(T_a) \) is defined as \( \{ k \in K_1 \mid T_a(k) \neq \text{ns} \} \). To define the composition, we need another meta-action called \( \text{fail} \). For all kinds \( K \), \( T_{\text{fail}}(K)(k) = \text{ns} \), and \( A_{\text{fail}}(K)(k) = \lambda v : k . () \). The action \( \text{fail} \) is composed with action \( a_1 \) to ensure the composite action has the appropriate kinding: the kinding of \( a_1 / a_2 \) must be identical to that of \( a_1 \) and \( a_2 \).

**Derived composition**

We use the combinators ; and * to define the derived composition \( \downarrow_K \) introduced in Chapter 1. \( a_1 \downarrow_K a_2 \) behaves like \( a_1 ; a_2 \) for the arguments in the facets designated by \( K \) and like \( a_1 * a_2 \) for arguments in all other facets. For actions \( a_1: K_1 \rightarrow K'_1 \) and \( a_2: K_2 \rightarrow K'_2 \), where \( K'_1 \leq K \leq K_2 \), we define \( a_1 \downarrow_K a_2: K_0 \rightarrow K'_0 \) as:

\[
a_1 \downarrow_K a_2 = (\text{forget}(K_0,K_1) ; a_1 ; \text{forget}(K'_1,K_3)) * (((\text{forget}(K_0,K_1) ; a_1 ; \text{forget}(K'_1,K_3)) * \text{forget}(K_0,K_4)) ; a_2)
\]

where \( K_3 = K'_1 - K, K_4 = K_2 - K, K_0 = K_1 \cap K_4, \) and \( K'_0 = K_3 \cap K'_2 \). \( K_3 \) represents the results produced by \( a_1 \) which are not given to \( a_2 \); these results are
merged with the results of $a_2$. $K_4$ represents the arguments required by $a_2$ which are not produced by $a_1$. The facets in $K_4$ flow vertically into $a_1 \Downarrow^K a_2$, as do the facets required by $a_1$.

The action $\text{forget}_{(K,K')}^k$ forgets unneeded facet information in its argument of kind $K$ and outputs only the information of kind $K'$. The $\text{forget}$ actions are actually explicit coercions used to make the actions polymorphic on the facets. The derived combinator hides them from the user. The flow of arguments through $a_1 \Downarrow^K a_2$ is shown in Figure 2.9.

Examples of composite actions

The typing and meaning functions for the action $\text{inc}A = \text{find} A \; ; \; \text{succ}$ are:

\[
T_{\text{inc}A} = T_{\text{succ}} \circ T_{\text{find} A}
\]

\[
A_{\text{inc}A} = \lambda d : D. \; \text{succ} (T_{\text{find} A}(d)) \circ \text{find} A(d)
\]
The typing function produces non-

-ns results for all types \( d \) that are subtypes of \( \{ A : \text{real} \} \). For example,

\[
T_{\text{incA}}(\{ A : \text{int}, B : \text{int} \}) = T_{\text{suc}}(T_{\text{find}} A(\{ A : \text{int}, B : \text{int} \})) = T_{\text{suc}}(\text{int}) = \text{int}
\]

Suppose we wish to apply \( \text{incA} \) to the record \( \{ A = 3, B = 4 \} \in \{ A : \text{int}, B : \text{int} \} \) in a context that expects a result of type \( \text{real} \). The following diagram shows us that it does not matter whether the record is coerced to the record \( \{ A = 3.0 \} \in \{ A : \text{real} \} \) before the action is applied, or whether the result \( \text{incA}(3) \in \text{int} \) is coerced to the value \( 4.0 \in \text{real} \):

\[
\begin{array}{ccc}
A_D[\{ A : \text{real} \}] & \xrightarrow{\text{incA} (\{ A : \text{real} \})} & A_F[\text{real}] \\
A_D[\{ A : \text{int}, B : \text{int} \} \leq \{ A : \text{real} \}] & \xrightarrow{\text{incA} (\{ A : \text{int}, B : \text{int} \})} & A_F[\text{int} \leq \text{real}] \\
A_D[\{ A : \text{int}, B : \text{int} \}] & \xrightarrow{\text{incA} (\{ A : \text{int}, B : \text{int} \})} & A_F[\text{int}] \\
\end{array}
\]

The result is the same in either case—the coercion is not needed by the action.

The typing and meaning functions for the action \( \text{copy} \ast \text{suc} \) are:

\[
\begin{align*}
T_{\text{copy}\ast\text{suc}} & = \lambda t : F . T_{\text{merge}(F,F)}(T_{\text{copy}}(t), T_{\text{suc}}(t)) \\
A_{\text{copy}\ast\text{suc}} & = \lambda t : F . \lambda v : t . A_{\text{merge}(F,F)}(T_{\text{copy}}(t), T_{\text{suc}}(t))(\text{copy } t v, \text{suc } t v)
\end{align*}
\]

Application of \( \text{copy} \ast \text{suc} \) to the value \( 3 \) of type \( \text{int} \) gives the results:

\[
\begin{align*}
T_{\text{copy}\ast\text{suc}}(\text{int}) & = T_{\text{merge}(F,F)}(T_{\text{copy}}(\text{int}), T_{\text{suc}}(\text{int})) \\
& = T_{\text{merge}(F,F)}(\text{int}, \text{int}) \\
& = \text{int} \times \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{copy} \ast \text{suc } 3 & = A_{\text{merge}(F,F)}(T_{\text{copy}}(\text{int}), T_{\text{suc}}(\text{int}))(\text{copy } 3, \text{suc } 3) \\
& = A_{\text{merge}(F,F)}(\text{int}, \text{int})(3,4) \\
& = (3,4)
\end{align*}
\]
The typing and meaning functions for the action \( \text{rebind} = \triangleright (\text{forget}_{FD,F} ; \text{bind } B) \) are defined as:

\[
T_{\text{rebind}} = \lambda(t, d) : FD . T_{\text{concat}}(d, T_{\text{bind } B} \circ T_{\text{forget}_{FD,F}}(t, d))
\]

\[
A_{\text{rebind}} = \lambda(t, d) : FD . \lambda(v, r) : t \times d .
\]

\[
\text{concat}(r, (\text{bind } B T_{\text{forget}_{FD,F}}(t, d)) \circ (\text{forget}_{FD,F}(t, d))(v, r))
\]

Application of the typing function to the types \text{int} and \{A : \text{int}, B : \text{bool}\} gives:

\[
T_{\text{rebind}}(\text{int}, \{A : \text{int}, B : \text{bool}\})
\]

\[
= T_{\text{concat}}(\{A : \text{int}, B : \text{bool}\}, T_{\text{bind } B} \circ T_{\text{forget}_{FD,F}}(\text{int}, \{A : \text{int}, B : \text{bool}\}))
\]

\[
= T_{\text{concat}}(\{A : \text{int}, B : \text{bool}\}, T_{\text{bind } B}(\text{int}))
\]

\[
= T_{\text{concat}}(\{A : \text{int}, B : \text{bool}\}, \{B : \text{int}\}_{\text{exactly}})
\]

\[
= \{A : \text{int}, B : \text{int}\}
\]

Application of \text{rebind} to the functional, declarative pair \((3, \{A = 5, B = \text{true}\})\) with types \text{int} and \{A : \text{int}, B : \text{bool}\} gives:

\[
\text{rebind} (\text{int}, \{A : \text{int}, B : \text{bool}\})(3, \{A = 5, B = \text{true}\})
\]

\[
= \text{concat}(\{A = 5, B = \text{true}\}, r_2)
\]

where \(r_2\) is equal to

\[
(\text{bind } B \text{ int } \circ \text{forget}_{FD,F}(\text{int}, \{A : \text{int}, B : \text{bool}\}))(3, \{A = 5, B = \text{true}\})
\]

\[
= \text{bind } B \text{ int } 3
\]

\[
= \{B = 3\}
\]

We obtain the result \(\text{concat}(\{A = 5, B = \text{true}\}, \{B = 3\}) = \{A = 5, B = 3\}\) in which the new binding of \(B\) to 3 in the result record overrides the binding of \(B\) to \text{true} in the argument record.
CHAPTER 3. TYPE INFERENCE

An action’s typing function can be quite complex. Our type inference algorithm annotates an action with a type scheme that encodes the same information as its typing function, but is much easier to read. For example, the action find $A \ast$ find $B$ has the typing function:

$$T_{\text{find } A \ast \text{find } B} = \lambda d : D \cdot T_{\text{merge}(F,F)}(T_{\text{find } A}(d), T_{\text{find } B}(d))$$

and the typing scheme:

$$\text{find } A \ast \text{find } B \in \{A: \theta_1, B: \theta_2\} \alpha \rightarrow \theta_1 \times \theta_2$$

The scheme states that when applied to any declarative facet argument whose type is a subtype of $\{A: \theta_1, B: \theta_2\}$, action find $A \ast$ find $B$ will produce a pair of values of types $\theta_1$ and $\theta_2$ as its result. Any instantiations of the type variables $\theta_1$ and $\theta_2$ with types from the functional facet, and the type variable $\alpha$ with bindings (other than $A$ and $B$) from the declarative facet is a legal (non-ns) typing for find $A \ast$ find $B$. For example, if we instantiate $\theta_1$ with real $\rightarrow$ bool, $\theta_2$ with int, and $\alpha$ with $\{C: \text{bool}\}$, application of the typing function to the instantiated scheme’s source gives the instantiated scheme’s target as its result:

$$T_{\text{find } A \ast \text{find } B}(\{A: \text{real }\rightarrow \text{bool}, B: \text{int}, C: \text{bool}\}) = (\text{real }\rightarrow \text{bool}) \times \text{int}$$
This property of an action's type scheme is called *soundness*.

An action's type scheme represents in schematic form the non-ns behavior of its typing function. Types which the typing function maps to ns are not instances of the source of the action's scheme. For example, $T_{\text{find} \ A \ \ast \text{find} \ B}(\{C: \text{bool}\}) = \text{ns}$; likewise, the type $\{C: \text{bool}\}$ is not an instance of the action's source scheme. $T_{\text{find} \ A \ \ast \text{find} \ B}$ produces non-ns results when applied to any record type with at least fields for A and B. Similarly, the source scheme $\{A: \theta_1, B: \theta_2\} \alpha$ unifies with any record type with at least A and B fields. Any type that the typing function maps to a non-ns result is an instance of the typing scheme. This property of an action's type scheme is called *completeness*.

Our type inference algorithm assigns a type scheme to an action expression that is both sound and complete with respect to the semantic model given in Chapter 2. Our algorithm thus infers the "best" type scheme for an action. Because the type inference and its soundness and completeness for each of the action combinators is defined independently, we shall sometimes refer to our type inference algorithm in the plural.

**ML-style, Unification-based Type Inference**

Language constructs in ML include function abstraction and application (i.e., the lambda calculus), and a set of polymorphic primitive operators [18]. It is the type constraints on these primitive operators and the declaration and use of identifiers which determine the types of an ML expression.

Milner defines a type inference algorithm $\mathcal{W}$ for a subset of ML. Algorithm $\mathcal{W}$ annotates an ML expression with a type scheme. Milner proves the soundness of
algorithm \( \mathcal{W} \), which he states as “whenever \( \mathcal{W} \) succeeds, it produces a well-typing.”

He states completeness as “whenever a well-typing exists, algorithm \( \mathcal{W} \) succeeds in finding one,” but does not prove it.

\( \mathcal{W} \) is defined in a syntax-directed manner: the type inference for an expression such as \( (E_1 E_2) \) is defined in terms of the type inference for its subexpressions \( E_1 \) and \( E_2 \). In order for the application \( (E_1 E_2) \) to be well-typed, \( E_1 \) should have a function type \( \tau_1 \rightarrow \tau_2 \) and \( E_2 \) should have type \( \tau_1 \). Because of the polymorphic operators, the types inferred for \( E_1 \) and \( E_2 \) may contain type variables. Function application in Milner’s system is thus polymorphic.

Application of a function to an argument requires that the argument’s type be equal to the domain type of the function. Two type schemes are made equal to one another by unification. Unification results in a substitution \( U \) that tells us how the type variables in the two type expressions must be filled in to make them equal. A substitution is a finite mapping of type variables to type expressions. If \( \tau \) is a type scheme and \( U \) is a substitution, then \( U(\tau) \) is the type scheme obtained by replacing each variable \( \theta \) in \( \tau \) by \( U(\theta) \). \( U(\tau) \) is called an instance of \( \tau \).

Robinson [35] shows there is an algorithm unify that takes a pair of expressions \( \tau \) and \( \sigma \) and produces a substitution \( U \) with the following properties:

(a) If \( \text{unify}(\tau, \sigma) \neq \text{failure} \), then \( U(\tau) = U(\sigma) \). \( U \) unifies \( \tau \) and \( \sigma \).

(b) If \( U_1 \) unifies \( \tau \) and \( \sigma \), then \( \text{unify}(\tau, \sigma) = U \neq \text{failure} \) and there exists a substitution \( U_2 \) such that \( U_1 = U_2 \circ U \).

The second property states that unify produces a most general or “best” substitution: type variables are filled in only enough to make the terms match.

Type inference for function application is defined in terms of unification:
\( W: \text{Expression} \rightarrow \text{Type-scheme} \)
\[
W(E_1 E_2) = \text{let } \tau_1 = W(E_1) \\
\text{let } \tau_2 = W(E_2) \\
\text{let } U = \text{unify}(\tau_1, \tau_2 \rightarrow \beta), \quad \text{where } \beta \text{ is new} \\
\text{in } U(\beta)
\]

(Note: we give a simplified description of algorithm \( W \) to focus on its features that are relevant to our type inference algorithm for action semantics.) Type inference for the expression \( \text{hd} [3,4] \) assigns the type scheme \( \alpha \text{ list} \rightarrow \alpha \) to the primitive \( \text{hd} \) and the type \( \text{int list} \) to the value \([3,4]\). Each occurrence of a primitive operator like \( \text{hd} \) is assigned a typing scheme with "fresh" (previously unused) type variables. The type schemes \( \alpha \text{ list} \rightarrow \alpha \) and \( \text{int list} \rightarrow \beta \) are then unified, causing the variables \( \alpha \) and \( \beta \) to both be instantiated with \( \text{int} \). The result of \( \text{hd} [3,4] \) is inferred to have type \( \text{int} \).

We limit our description of type inference in ML to primitive operators and function application. Our definition of type inference for primitive actions and action combinators is based on the type inference for these constructs. Milner points out that polymorphic primitive operators appear to exist in all programming languages in such forms as assignment, pairing and tupling, list processing operations, and function application. Indeed, if we consider the action semantics metalanguage as a programming language itself, we have a language consisting entirely of such operators. And further, the language is used to define other languages.

The kind of polymorphism in ML is parametric (also called "structural"). ML-style type inference has been extended to inclusion polymorphism in the form of subtypes and records by a number of researchers [10, 11, 15, 20, 31, 41, 42]. Chapter 6 discusses these extensions. Our type inference algorithm for action semantics also
extends ML-style type inference with subtypes and records, and additionally adds a form of ad-hoc polymorphism.

Type Schemes

We now define the language of type schemes for each of the facets and the type schemes for the set of primitive actions. Our type schemes are based on those introduced by Milner for ML and its extensions to subtypes by Mitchell [20] and records by Wand [41].

Facets

Figure 3.1 gives the syntax of the typing schemes for each facet. Type variables are represented by $\theta$, $\Delta$, $\alpha$, $\sigma$, and $\iota$. $\theta$ stands for an arbitrary proper type in the functional facet and may be instantiated to any type scheme generated by $\nu$. $\Delta$ is a field variable [31, 42]; it represents unknown status about the “presence” or “absence” of an identifier in a record scheme. $\alpha$ is a row variable [41]; it indicates unknown status about all identifiers not mentioned in the index set of the record type scheme. Field and row variables are used for “bookkeeping” during type inference as we shall see later. The present-tag on a field’s type scheme is merely syntactic sugar to make it clear that we cannot unify a “present” $\theta$-field with an “absent” absent-field. The absent-field is used in conjunction with exactly-tagged record schemes. It is sometimes convenient to treat two records as having the same index set and the use of absent-fields allows us to do this.

As an example, the scheme $\{A : \text{present} (\theta), B : \Delta, C : \text{absent}\}_{\text{exactly}}$ represents the family of record types that have an $A$-field, may or may not have a $B$-field,
\( p \in \text{Primitive-type} \)
\( v \in \text{Functional-type-scheme} \)
\[
p ::= \text{nat} \mid \text{int} \mid \text{real} \mid \text{bool}
\]
\[
v ::= p \mid v_1 \times v_2 \mid v_1 \rightarrow v_2 \mid \emptyset \mid (l : v)\text{var}
\]
\( i \in \text{Identifier} \)
\( f \in \text{Field} \)
\( d \in \text{Record-type-scheme} \)
\( r \in \text{Row} \)
\[
f ::= \text{absent} \mid \Delta \mid \text{present}(v)
\]
\[
r ::= \alpha \mid \alpha_{\text{exactly}} \mid \text{exactly}
\]
\[
d ::= \{i : f_i\}_{i \in l}\text{r}
\]
\( l \in \text{Location} \)
\( m \in \text{Cell} \)
\( l \in \text{Locn} \)
\( s \in \text{Storage-type-scheme} \)
\[
l ::= \epsilon \mid l
\]
\[
m ::= \text{undef} \mid v
\]
\[
s ::= (l : m_i)_{i \in l}\sigma
\]
\( e \in \text{Constraint} \)
\( c \in \text{Constraints} \)
\[
e ::= v_1 \leq v_2 \mid s_1 \leq s_2
\]
\[
c ::= c_1 \text{ and } c_2 \mid e
\]
\( S \in \text{Typing-scheme} \)
\[
S ::= 1 \mid v \mid d \mid s \mid v \times d \mid v \times s \mid d \times s \mid v \times d \times s
\]

Figure 3.1. Syntax of Type Schemes
definitely do not have a \( C \)-field, may or may not have other fields beyond \( A \), \( B \), and \( C \), and finally has an exact number of fields.

We assume that a row variable \( \alpha \) has as its "domain of use" its index set \( I \) and can only be instantiated with fields whose labels are not in \( I \) \([31, 40, 42]\). We also assume that all occurrences of a row variable \( \alpha \) appear in the same context, i.e., have the same index set on their associated record type scheme.

\( \sigma \) is a row variable for storage types. To distinguish them from the row variables for record type schemes, we will call them storage variables. The same assumptions on row variables apply to storage variables. Storage cells also have associated bookkeeping information. The \texttt{undef} cell indicates that its location is undefined in the storage type. Again, as for record type schemes, this information is needed to ensure that all occurrences of a storage variable have the same context. The type variable \( \iota \) may appear in a storage type scheme to indicate unknown status about the exact location of the cell. This allows us to do inference on the type of value stored in a cell without mention of the cell's location.

We will often abbreviate a record type scheme by omitting the index set for its identifiers when it can be inferred. For example, the scheme \( \{i : \Delta_i, \, j : f_j\}_\alpha \) abbreviates the scheme \( \{i : \Delta_i, \, j : f_j\}_{i \in I, \, j \in J} \alpha \), where \( I \cap J = \emptyset \).

**Primitive actions**

An action has a typing scheme of the form \( S_1 \vdash S_2 \text{ if } C \), where \( S_1 \) and \( S_2 \) are facet typing schemes, and \( C \) is a set of constraints. When there are no constraints, i.e., \( C = \text{true} \), we omit the set of constraints from the scheme. We will sometimes use a script letter \( \mathcal{S} \) to refer to an action's scheme. Figure 3.2 gives the syntax of
ACTION SCHEME

put (n : t₀)  1 ⊢ t₀
succ θ ⊢ θ if θ ≤ real
pass { }α ⊢ { }α
bind i θ ⊢ {i : present(θ)} exactly
find i {i : present(θ)}α ⊢ θ
copy θ ⊢ θ
fst θ₁ × θ₂ ⊢ θ₁
snd θ₁ × θ₂ ⊢ θ₂
add_int θ₁ × θ₂ ⊢ int if θ₁ ≤ int and θ₂ ≤ int
add_real θ₁ × θ₂ ⊢ real if θ₁ ≤ real and θ₂ ≤ real
skip ⟨⟩σ ⊢ ⟨⟩σ
alloc₀ ⟨l : undef⟩σ ⊢ ⟨l : t₀⟩var × ⟨l : t₀⟩σ where l = newlocn()
contents ⟨l : θ⟩var × ⟨l : θ⟩σ ⊢ θ
update ⟨(l : θ)var × θ'⟩ × ⟨l : θ⟩σ ⊢ ⟨l : θ⟩σ if θ' ≤ θ

Figure 3.2. Typing Schemes for Primitive Actions

type schemes for the primitive and bridging actions.

When reading a typing scheme, there is implicit universal quantification on each
type variable, provided its instantiation will not make the constraints false. For
example, the typing scheme succ ∈ θ ⊢ θ if θ ≤ real is read as “for all instantiations
of θ such that θ is a subtype of real, succ has type θ → θ.” The scheme abbreviates
the typing function λt ∈ F. if t ≤ real then t else ns. The next section discusses
the correctness of an action scheme with respect to the model.

Soundness and completeness

We would like to know whether the typing schemes given in Figure 3.2 correctly
represent the behaviors of the typing functions for the actions. We are interested
in two properties of the typing scheme for a given action: its soundness and its
completeness with respect to the action's typing function. Both properties key on instantiations into the typing scheme that make the constraints hold true.

The general notion of an instantiation into a typing scheme is represented by a substitution. We are interested in substitutions that will allow the scheme's set of constraints to be satisfied. For example, the substitution $U = [\theta_1 \leftarrow \theta_2 \times \theta_3]$ does not allow the constraint $\theta_1 \leq \text{real}$ to be satisfied because there is no way to instantiate $\theta_2$ and $\theta_3$ to make the constraint $U(\theta_1 \leq \text{real}) = \theta_2 \times \theta_3 \leq \text{real} \text{ hold in our subtype ordering}$. The constraint $U(\theta_1 \leq \text{real})$ is not satisfiable. Satisfaction of constraints will be addressed in the next chapter.

We say that a substitution $U$ is ground with respect to scheme $S_1 \Rightarrow S_2$ if $C$ if no type variables remain in the scheme after the substitution is applied. For example, the substitution $U = [\theta_4 \leftarrow \text{int}][\theta_5 \leftarrow \text{bool}]$ is ground with respect to the scheme $S = \theta_5 \Rightarrow \theta_6 \text{ if } \theta_4 \leq \text{real}$ because no type variables remain in the scheme $U(S)$.

When the scheme we are concerned with is clear, we will simply refer to $U$ as a ground substitution. The result of applying a ground substitution to a type scheme is a ground type scheme. The next section discusses the translation of a ground type scheme into a type.

We have the following definitions for soundness and completeness:

**Soundness**  A typing scheme $S_1 \Rightarrow S_2$ if $C$ is sound for action $a: K_1 \Rightarrow K_2$ if and only if for all types $k \in K_1$, for all ground substitutions $U$, $U(S_1) = k \neq \text{ns}$ and $U(C)$ holds true imply that $T_a(k) = U(S_2)$.

**Completeness** A typing scheme $S_1 \Rightarrow S_2$ if $C$ is complete for action $a: K_1 \Rightarrow K_2$ if and only if for all types $k \in K_1$, $T_a(k) \neq \text{ns}$ implies there exists a ground substitution $U$ such that $U(S_1) = k$ and $U(C)$ holds true.
The proofs of soundness and completeness for the primitive type schemes are straightforward so we omit them. The key point is showing that the subtyping constraints in an action's typing function are "checked" by the action's typing scheme. For example, recall that the typing function for contents is \( T_{\text{contents}}(t', s) = \text{if } t' = (\ell:t) \text{var and } s \leq (\ell:t) \text{ then } t \text{ else } \text{ns} \). The subtyping on storage is enforced by contents's source scheme on the imperative facet: to produce a non-ns typing, ground substitutions into the scheme \( (\iota:\theta)\sigma \) must instantiate \( \iota \) to a location, say \( \ell \), \( \theta \) to a type in the functional facet, say \( t \), and \( \sigma \) to a set of storage cells with locations other than \( \ell \), say \( (\ell_1:t_1, \ldots, \ell_n:t_n) \). Application of this substitution to the storage scheme results in the type \( (\ell:t, \ell_1:t_1, \ldots, \ell_n:t_n) \), which is a subtype of \( (\ell:t) \).

**Equivalence rewrite rules** Ground type schemes in the declarative and imperative facets differ syntactically from types in these facets. Record type schemes and storage type schemes contain extra bookkeeping information in each field that is not present in their corresponding types. We use rewrite rules to translate a type scheme into a type.

A field variable in a record scheme can be instantiated to either **absent** or **present** \( t \), for some type \( t \). "absent" fields don't exist as types, but they appear in "exactly" record type schemes to indicate that the field is not present in the record. We eliminate them by the following rule:

\[
(E-i) \quad \{ \ldots I: \text{absent} \ldots \} \text{exactly} \Rightarrow \{ \ldots \} \text{exactly}
\]

The "present" tag on a field value is eliminated by the rules:

\[
(E-\text{ii}) \quad \{ \ldots I: \text{present}(t) \ldots \} \text{exactly} \Rightarrow \{ \ldots I: t \ldots \} \text{exactly}
\]

\[
(E-\text{iii}) \quad \{ \ldots I: \text{present}(t) \ldots \} \Rightarrow \{ \ldots I: t \ldots \}
\]
Storage type schemes may contain "undefined" cells which are not present in the typing system. These tags are needed in the type inference algorithm by the function newlocn. These bookkeeping cells are eliminated from a ground storage type scheme by the rule:

$$(E-iv) \quad \langle \ldots \ell \; \text{undef} \ldots \rangle \Rightarrow \langle \ldots \ldots \rangle$$

Note that some of our "ground schemes" are not actually "schemes". For example, we allow the row variable $\alpha$ in the record scheme $\{A: \text{present}((\text{int}))\alpha$ to be instantiated to $\{\}$. But the resulting ground scheme, $\{A: \text{present}((\text{int}))\}$, is not generated by our syntax of schemes.

**Type Inference for Action Combinators**

We now present the type inference algorithms for each of the action combinators and prove soundness and completeness. We introduce functions used by the type inference algorithms as we go along.

**Sequential composition**

For actions $a_1^{S_1}$ and $a_2^{S_2}$, we define the type inference for sequential composition as $(a_1; a_2)^{semi(S_1, S_2)}$, where

$semi(S_1 \vdash S_2 \text{ if } C, S'_1 \vdash S'_2 \text{ if } C') =$

$\begin{align*}
U(S_1) \vdash U(S'_1) \text{ if } U(C \text{ and } C'), \\
\text{if } U = \text{unify}(S_2, S'_1) \neq \text{failure} \text{ and } \text{is-satisfiable}(U(C \text{ and } C'))
\end{align*}$

$\begin{align*}
\text{ns } \vdash \text{ns if false, otherwise}
\end{align*}$
The type inference uses the function \textit{is-satisfiable}: \(\text{Constraints} \rightarrow \text{Bool}\), which states whether there exists a ground substitution \(U_0\) for a set of constraints \(C\) such that \(U_0(C)\) holds true. The algorithms for checking the satisfaction of a set of constraints are presented in the next chapter.

\textit{semi} depends on the unification of the target scheme for \(a_1\) and the source scheme for \(a_2\). ML-style unification is used for the type schemes in the functional facet. The algorithms for the unification of record and storage type schemes are given in Figures 3.3 and 3.4.

As an example, the unification of \(\{A: \text{present}(\theta_2), B: \text{present(\text{bool})}\}_{\text{exactly}}\) with \(\{A: \Delta_1, C: \text{absent}\}_{\text{exactly}}\) produces the substitution \([\alpha \mapsto \{B: \text{present(\text{bool})}\}] [\Delta_1 \mapsto \text{present}(\theta_2)]\) and the scheme \(\{A: \text{present}(\theta_2), B: \text{present(\text{bool})}, C: \text{absent}\}_{\text{exactly}}\).

The unification of storage types needs some explanation. \textit{unify-storage} does not distinguish between locations \(\ell\) and location type variables \(\iota\). When unifying the schemes \(\langle \ell_1: \text{int}, \ell_2: \text{int} \rangle \sigma_1\) and \(\langle \iota_2: \text{int}, \iota_4: \text{int} \rangle \sigma_2\), we do not have enough information to know whether to instantiate either of \(\iota_3\) or \(\iota_4\) to one of the locations. Instead, unification produces the substitution \([\sigma_1 \mapsto \langle \iota_3: \text{int}, \iota_4: \text{int} \rangle \sigma_3] [\sigma_2 \mapsto \langle \ell_1: \text{int}, \ell_2: \text{int} \rangle \sigma_3]\).

Location type variables may only be unified with other location type variables or instantiated to locations when they appear as part of a variable type scheme \(\langle \ell: v \rangle \text{var}\). This has the effect that two locations appearing in the same storage type scheme may become equivalent as the type inference on an action expression progresses. We thus can have storage schemes with duplicate locations. This does not present any problems as we can rewrite a scheme \(\langle l: m, l: m' \rangle \sigma\) as \(\langle l: m \rangle \sigma\), provided \(m = m'\), so we will simply ignore this anomaly.
unify-records: Record × Record → Substitution
unify-records(d_1, d_2) =
  let \( I = \text{fields-in}(d_1) \cup \text{fields-in}(d_2) \)
  let \( U_1, d'_1 = \text{extend-record}(I \setminus \text{fields-in}(d_1), d_1) \)
  let \( U_2, d'_2 = \text{extend-record}(I \setminus \text{fields-in}(d_2), d_2) \)
  let \( \{\text{fields}_1\} r_1 = d'_1 \)
  let \( \{\text{fields}_2\} r_2 = d'_2 \)
  let \( U_3 = \text{unify-row-vars}(r_1, r_2) \)
  let \( U_4 = \text{unify-fields}(I, \text{fields}_1, \text{fields}_2) \)
in \( U_4 \circ U_3 \circ U_2 \circ U_1 \)

extend-record: Labels × Record → Substitution × Record
extend-record(J, \{i : f_i\}_\text{exactly}) = [\], \{i : f_i, j : \text{absent}\}_\text{exactly}
extend-record(J, \{i : f_i\}_\alpha) = [\alpha \mapsto \{j : \Delta_j\}_\beta], \{i : f_i, j : \Delta_j\}_\beta
extend-record(J, \{i : f_i\}_\alpha\_\text{exactly}) = [\alpha \mapsto \{j : \Delta_j\}_\beta], \{i : f_i, j : \Delta_j\}_\beta\_\text{exactly}

unify-row-vars: Row × Row → Substitution
unify-row-vars(r_1, r_2) = \text{cases}(r_1, r_2) \text{ of}
  \text{(exactly, exactly)} \Rightarrow [\]
  (exactly, \alpha) or (\alpha, exactly) \Rightarrow [\alpha \mapsto \{}_\text{exactly}
  (exactly, \alpha\_\text{exactly}) or (\alpha\_\text{exactly}, exactly) \Rightarrow [\alpha \mapsto \{}]
  (\alpha, \beta) \Rightarrow [\alpha \mapsto \delta][\beta \mapsto \delta]
  (\alpha\_\text{exactly}, \beta) \Rightarrow [\alpha \mapsto \delta\_\text{exactly}][\beta \mapsto \delta]
  (\alpha\_\text{exactly}, \beta\_\text{exactly}) \Rightarrow [\alpha \mapsto \delta][\beta \mapsto \delta]

unify-fields: Label-list × Record × Record → Substitution
unify-fields(nil, d_1, d_2) = [\]
unify-fields(\ell :: i, d_1, d_2) =
  let \( U_1 = \text{unify-fields}(\ell, d_1, d_2) \)
  let \( U_2 = \text{unify}(U_1(d_1,i), U_1(d_2,i)) \)
in \( U_2 \circ U_1 \)

Figure 3.3. Unification of Record Type Schemes
unify-storage: Storage-type-scheme × Storage-type-scheme → Substitution

unify-storage(s₁, s₂) =
  let I = locations-in(s₁) ∩ locations-in(s₂)
  let U = unify-cells(I, s₁, s₂)
  let \( \langle i : m_i, j : m_j \rangle \sigma_1 = U(s_1) \)
  let \( \langle i : m_i, k : m_k \rangle \sigma_2 = U(s_2) \)
  in \( \sigma_1 \mapsto (k : m_k) \sigma_1 \sigma_2 \mapsto (j : m_j) \sigma_1 \circ U \)

unify-cells: Location-list × Storage-scheme × Storage-scheme → Substitution

unify-cells(nil, s₁, s₂) = []
unify-cells(i :: ℓ, s₁, s₂) =
  let U₁ = unify-cells(ℓ, s₁, s₂)
  let U₂ = unify(U₁(s₁.i), U₁(s₂.i))
  in U₂ \circ U₁

Figure 3.4. Unification of Storage Type Schemes

An example  We now show the type inference for the action pass ; find B.

First, schemes are inferred for the primitive actions:

\[
\text{pass} \quad \in \quad \{ \} \alpha_1 \mapsto \{ \} \alpha_1
\]
\[
\text{find } B \quad \in \quad \{ B : \text{present}(\theta_1) \} \alpha_2 \mapsto \theta_1
\]

Each action scheme is given distinct type variables. Next, type inference for ; unifies \( \{ \} \alpha_1 \) and \( \{ B : \text{present}(\theta_1) \} \alpha_2 \) to produce the substitution:

\[
U = [\alpha_1 \mapsto \{ \} \alpha_3][\alpha_3 \mapsto \alpha_4][\alpha_2 \mapsto \alpha_4][\Delta_1 \mapsto \text{present}(\theta_1)]
\]

There are no constraints, so \( U \) is applied to the source and target typing schemes to obtain the scheme pass; find B \( \in \{ B : \text{present}(\theta_1) \} \alpha_4 \mapsto \theta_1 \).

As another example, we show the type inference for find B; bind A. The type
schemes for the primitive actions are:

\[
\text{find } B \in \{ B : \text{present}(\theta_1) \} \alpha_1 \vdash \theta_1 \\
\text{bind } A \in \theta_2 \vdash \{ A : \text{present}(\theta_2) \} \text{ exactly}
\]

\(\theta_1\) and \(\theta_2\) are unified to a new type variable \(\theta_3\), and the type inference produces the following typing scheme for the action:

\[
\text{find } B; \text{bind } A \in \{ B : \text{present}(\theta_3) \} \alpha_1 \vdash \{ A : \text{present}(\theta_3) \} \text{ exactly}
\]

**Soundness and completeness of sequential composition** We now show soundness and completeness for semi. Recall that the typing function for the combinator "\(;\)" is \(T_a_1; a_2 = T_a_2 \circ T_a_1\).

**Proposition 3.1** If scheme \(S_1 = S_1 \vdash S_2\) if \(C\) is sound for action \(a_1 : K_1 \rightarrow K_2\), and scheme \(S_2 = S'_1 \vdash S'_2\) if \(C'\) is sound for action \(a_2 : K_2 \rightarrow K'_2\), then scheme \(\text{semi}(S_1, S_2)\) is sound for action \(a_1; a_2\).

**Proof** Let \(\text{semi}(S_1, S_2) = S \vdash S'\) if \(D\). Let \(k \in K_1\), and let \(U_0\) be a ground substitution such that \(U_0(S) = k \neq \text{ns}\) and \(U_0(D)\) holds true. Then \(U = \text{unify}(S_1, S'_1) \neq \text{failure}\), since \(k \neq \text{ns}\). Thus, \(k = U_0(U_0(S_1))\). Since \(U_0(D) = U_0(U(C \text{ and } C'))\) holds true, \(U_0(U(C) = \text{true} \text{ and } U_0(U(C') = \text{true}\).

By the soundness of \(a'_1\), \(T_{a_1}(k) = U_0(U(S_2)) = k' = U_0(U(S'_1))\), since \(U(S_2) = U(S'_1)\). If \(k' \neq \text{ns}\), by the soundness of \(a'_2\), \(T_{a_2}(k') = U_0(U(S'_2)) = U_0(S')\). Thus, \(T_{a_1; a_2}(k) = T_{a_2}(T_{a_1}(k)) = U_0(S')\). \(\Box\)

**Lemma 3.1** Let \(S_1 = S_1 \vdash S_2\) if \(C\) be the scheme for action \(a_1 : K_1 \rightarrow K_2\), and let \(S_2 = S'_1 \vdash S'_2\) if \(C'\) be the scheme for action \(a_2 : K_2 \rightarrow K'_2\). For all types \(k \in K_1\) and
k' ∈ K₂, if there exist ground substitutions U₁ and U₂ such that U₁(S₁) = k, U₁(C)
holds, and U₁(S₂) = k' ≠ ns, and U₂(S₂') = k', and U₂(C') holds, then there exists a
ground substitution U₀ such that U₀(U(S₁)) = k and U₀(U(C and C')) holds, where
U = unify(S₂, S₂').

Proof Since there are no variables used in common by S₁ and S₂, U₁ ∪ U₂ is a
unifier of S₂ and S₂', and U₁ U₂(C and C') holds. Hence, there exists a substitution
U' such that U' = unify(S₂, S₂') ≠ failure. Since unification produces a most general
unifier, there must exist a ground substitution U₀ such that U₀ o U' = U₁ ∪ U₂. U₀
is the desired substitution. □

Proposition 3.2 If scheme S₁ = S₁ ⊢ S₂ if C is sound and complete for action
a₁: K₁ ⊢ K₂, and scheme S₂ = S₂' ⊢ S₂' if C' is complete for action a₂: K₂ ⊢ K₂',
then (a₁; a₂) is complete.

Proof Say that T_{a₁,a₂}(k) = k'' ≠ ns, where T_{a₁}(k) = k' ≠ ns, by strictness, and
T_{a₂}(k') = k''. By the completeness of S₁, there exists a ground substitution U₁ such
that U₁(S₁) = k and U₁(C) holds. By the soundness of S₁, U₁(S₂) = k'. By the
completeness of S₂, there exists a ground substitution U₂ such that U₂(S₂') = k' and
U₂(C') holds. Now by Lemma 3.1, the result follows. □

Parallel composition

For actions a₁S₁ and a₂S₂, we define the type inference for parallel composition as
(a₁ * a₂)_{star(S₁,S₂)}, where
\[ \text{star}\left(S_1 \sim S_2 \text{ if } C, S'_1 \sim S'_2 \text{ if } C'\right) = \\
U' \circ U(S_1) \sim S' \text{ if } U' \circ U(C \text{ and } C') \]

where \( U = \text{unify}(S_1, S'_1) \neq \text{failure} \)
and \( U', S' = S_{\text{merge}}(U(S_2), U(S'_2)) \neq \text{failure} \)
and \( \text{is-satisfiable}(U' \circ U(C \text{ and } C')) \)

\[ \text{ns} \leftarrow \text{ns if false, otherwise} \]

Parallel composition depends on the type inference defined by \( S_{\text{merge}} \). When merging functional facet schemes, ordinary tupling occurs: \( S_{\text{merge}}(F, F)(v_1, v_2) \) infers the product type scheme \( v_1 \times v_2 \). When merging records, the records are unioned if they have no fields in common. \( S_{\text{merge}}(D, D) \) is defined in Figure 3.5. The type inference must verify that the index sets of the two records are disjoint and that both record type schemes are “exactly”. Both schemes are extended to have identical sets of labels and \( \text{exactly-tags} \). \( \text{resolve-fields} \) ensures that labels with a \textit{present}-field in one record have an \textit{absent}-field in the other.

The key to keeping the inference simple is the omission of the cases in functions \( \text{resolve-row-vars} \) and \( \text{resolve-fields} \). There is no avoiding complex record constraints when unioning record schemes with two distinct row variables, like \{\ldots\}\alpha \) and \{\ldots\}\beta, or two distinct field variables, like \{A : \Delta_1\}_{\text{exactly}} \) and \{A : \Delta_2\}_{\text{exactly}} \) [42]. (One of the \( \Delta \)-variables must be \textit{absent}, but we do not know which one.) In a later section, we will show that the typing schemes for action expressions have properties that make these omitted cases impossible in our system.
\( S_{\text{merge}}(D,D) \) : Declarative-facet-scheme \( \times \) Declarative-facet-scheme
\( \text{→ Substitution} \times \text{Declarative-facet-scheme} \)

\( S_{\text{merge}}(d_1,d_2) = \)
\begin{align*}
\text{let } I &= \text{fields-in}(d_1) \cup \text{fields-in}(d_2) \\
\text{let } U_1, d'_1 &= \text{extend-record}(I \setminus \text{fields-in}(d_1), d_1) \\
\text{let } U_2, d'_2 &= \text{extend-record}(I \setminus \text{fields-in}(d_2), d_2) \\
\text{let } U_3, d &= S_{\text{union}}(I, d'_1, d'_2) \\
\text{in } U_3 \circ U_2 \circ U_1, d
\end{align*}

\( S_{\text{union}}: \text{Labels} \times \text{Record} \times \text{Record} \text{→ Substitution} \times \text{Record} \)

\( S_{\text{union}}(I, \{\text{fields}_1\}\{\text{row}_1\}, \{\text{fields}_2\}\{\text{row}_2\}) = \)
\begin{align*}
\text{let } U_1, \text{row} &= \text{resolve-row-vars}(\text{row}_1, \text{row}_2) \\
\text{let } U_2, \text{fields} &= \text{resolve-fields}(I, \text{fields}_1, \text{fields}_2) \\
\text{in } U_2 \circ U_1, \{\text{fields}\}\text{row}
\end{align*}

\( \text{resolve-row-vars}: \text{Row} \times \text{Row} \text{→ Substitution} \)

\( \text{resolve-row-vars}(r_1, r_2) = \text{cases } (r_1, r_2) \text{ of} \)
\begin{align*}
(\text{exactly},\text{exactly}) &\Rightarrow [\cdot], \text{exactly} \\
(\text{exactly}, a\text{exactly}) \text{ or } (a\text{exactly}, \text{exactly}) &\Rightarrow [\cdot], a\text{exactly} \\
(\text{exactly}, a) \text{ or } (a, \text{exactly}) &\Rightarrow [a \mapsto \beta\text{exactly}], \beta\text{exactly} \\
(a,a) &\Rightarrow [a \mapsto \{\cdot}\text{exactly}], \text{exactly} \\
(a\text{exactly}, a\text{exactly}) &\Rightarrow [a \mapsto \{\cdot}\text{exactly}], \text{exactly} \\
(a\text{exactly}, \beta\text{exactly}) &\Rightarrow \text{do not appear in our system}
\end{align*}

\( \text{resolve-fields}: \text{Labels} \times \text{Record} \times \text{Record} \text{→ Substitution} \)

\( \text{resolve-fields}(\text{nil}, d_1, d_2) = [\cdot], \{\cdot\} \)

\( \text{resolve-fields}(i :: \ell, d_1, d_2) = \)
\begin{align*}
\text{let } U_1, d &= \text{resolve-fields}(\ell, d_1, d_2) \\
\text{let } U_2, f &= \text{cases } (U_1(d_1.i), U_1(d_2.i)) \text{ of} \\
&\quad (\text{absent}, f) \text{ or } (f, \text{absent}) \Rightarrow [\cdot], f \\
&\quad (\Delta, \Delta) \Rightarrow [\Delta \mapsto \text{absent}], \text{absent} \\
&\quad (\Delta, \text{present}(v)) \text{ or } (\text{present}(v), \Delta) \Rightarrow [\Delta \mapsto \text{absent}], \text{present}(v) \\
&\quad (\text{present}(v), \text{present}(v')) \Rightarrow \text{failure} \\
&\quad (\Delta, \Delta') \Rightarrow \text{do not appear in our system}
\end{align*}
\in U_2 \circ U_1, \{i : f\}@d

Figure 3.5. Type Inference for Merge on the Declarative Facet
An example As an example, we show the type inference for \((\text{find } B; \text{bind } A) \ast \text{pass}\). The type inference for the arguments to \(\ast\) was shown in the last section:

\[
\begin{align*}
\text{find } B; \text{bind } A & \in \{B : \text{present}(\theta_3)\} \alpha_1 \Rightarrow \{A : \text{present}(\theta_3)\} \text{exactly} \\
\text{pass} & \in \{\} \alpha_2 \Rightarrow \{\} \alpha_2
\end{align*}
\]

The type inference for \(\ast\) unifies \(\{B : \text{present}(\theta_3)\} \alpha_1\) and \(\{\} \alpha_2\) to produce substitution \(U_1 = [\alpha_2 \leftarrow \{B : \Delta_1\} \alpha_3][\alpha_3 \leftarrow \alpha_4][\alpha_1 \leftarrow \alpha_4][\Delta_1 \leftarrow \text{present}(\theta_3)]\). Next, \(S_{\text{merge}(D,D)}\) is applied to the two target schemes \(\{A : \text{present}(\theta_3)\} \text{exactly}\) and \(\{B : \text{present}(\theta_3)\} \alpha_4\) to produce the substitution \(U_2 = [\alpha_4 \leftarrow \{A : \Delta_2\} \alpha_5 \text{exactly}][\Delta_2 \leftarrow \text{absent}]\), and the new target scheme \(\{A : \text{present}(\theta_3), B : \text{present}(\theta_3)\} \alpha_5 \text{exactly}\). The substitution \(U_2 \circ U_1\) is applied to the source scheme and we obtain the following typing scheme for the action:

\[
\begin{align*}
\text{find } B; \text{bind } A \ast \text{pass} & \in \{B : \text{present}(\theta_3), A : \text{absent}\} \alpha_5 \text{exactly} \\
& \Rightarrow \{A : \text{present}(\theta_3), B : \text{present}(\theta_3)\} \alpha_5 \text{exactly}
\end{align*}
\]

Soundness and completeness of merge \(S_{\text{merge}}\) is a “scheme constructor” in a sense: when applied to a pair of schemes \(S_2\) and \(S'_2\), with kindings \(K_2\) and \(K'_2\), \(S_{\text{merge}(K_2,K'_2)}\) produces a new target scheme \(S\), together with a substitution \(U\). The substitution must be applied to the source schemes \(S_1\) and \(S'_1\) to restrict them to exactly those instantiations that will produce a non-ns target type. Because of this substitution, we have slightly different formulations of soundness and completeness for \(S_{\text{merge}}\) than we have for the other actions. We state these properties for \(S_{\text{merge}}\) on the declarative facet.
Definition 3.1 \( S_{\text{merge}}(D,D) \) is sound iff for all declarative facet schemes \( d_1 \) and \( d_2 \), for all record types \( d \) and \( d' \), if \( S_{\text{merge}}(D,D)(d_1, d_2) = U, d_3 \neq \text{failure} \), then for all ground substitutions \( U_0, U_0(U(d_1)) = d \neq \text{ns} \) and \( U_0(U(d_2)) = d' \neq \text{ns} \) imply \( T_{\text{merge}}(D,D)(d, d') = U_0(d_3) \).

Definition 3.2 \( S_{\text{merge}}(D,D) \) is complete iff for all declarative facet schemes \( d_1 \) and \( d_2 \), if \( S_{\text{merge}}(D,D)(d_1, d_2) = U, d_3 \neq \text{failure} \), then for all ground substitutions \( U_0 \) such that \( T_{\text{merge}}(D,D)(U_0(d_1), U_0(d_2)) \neq \text{ns} \), there exists a ground substitution \( U'_0 \) such that \( U_0 = U'_0 \circ U \).

Our notion of "completeness" for \( S_{\text{merge}} \) is actually a weakness property on the substitution returned by \( S_{\text{merge}}(D,D) \). The proofs of soundness and completeness for \( S_{\text{merge}} \) are omitted.

Soundness and completeness of parallel composition  We now show soundness and completeness for \( \ast \). Recall that the typing function for the combinator \( \ast \) is \( T_{a_1 \ast a_2}(k) = T_{\text{merge}}(K_2, K_1')(T_{a_1}(k), T_{a_2}(k)) \).

Proposition 3.3 If scheme \( S_1 = S_1 \twoheadrightarrow S_2 \) if \( C \) is sound for action \( a_1:K_1 \twoheadrightarrow K_2 \), and scheme \( S_2 = S_2' \twoheadrightarrow S_2' \) if \( C' \) is sound for action \( a_2:K_1 \twoheadrightarrow K_2' \), then scheme \( \ast(S_1, S_2) \) is sound for action \( a_1 \ast a_2 \).

Proof Let \( \ast(S_1, S_2) = S \twoheadrightarrow S' \) if \( D \), let \( k \in K_1 \), and let \( U_0 \) be a ground substitution such that \( U_0(S) = k \neq \text{ns} \) and \( U_0(D) \) holds true. Then \( U = \text{unify}(S_1, S_1') \neq \text{failure} \), since \( k \neq \text{ns} \). Similarly, \( U', S' = S_{\text{merge}}(U(S_2), U(S_2')) \neq \text{failure} \). Thus, \( k = U_0(U' \circ U(S_1)) \). Since \( U_0(D) = U_0(U' \circ U(C' \text{ and } C')) \) holds true, \( U_0(U' \circ U(C')) = \text{true} \) and \( U_0(U' \circ U(C')) = \text{true} \).
By the soundness of $a_1^{S_1}$, $T_{a_1}(k) = U_0(U' \circ U(S_2)) = k'$. By the soundness of $a_2^{S_2}$, $T_{a_2}(k) = U_0(U' \circ U(S'_2)) = k''$. By the soundness of $S_{merge}$, $T_{merge(K_2,K_1')}(k',k'') = U_0(S')$. Thus, $T_{a_1*a_2}(k) = T_{merge(K_2,K_1')}(T_{a_1}(k),T_{a_2}(k)) = U_0(S')$. □

Lemma 3.2 Let $S_1 = S_1 \rightsquigarrow S_2$ if $C$ be the scheme for action $a_1: K_1 \rightsquigarrow K_2$, and let $S_2 = S'_1 \rightsquigarrow S'_2$ if $C'$ be the scheme for action $a_2: K_1 \rightsquigarrow K_2'$. For all types $k \in K_1$, if there exist ground substitutions $U_1$ and $U_2$ such that $U_1(S_1) = k \neq \text{ns}$ and $U_1(C)$ holds, and $U_2(S'_1) = k$ and $U_2(C')$ holds, then there exists a ground substitution $U_0$ such that $U_0(U(S_1)) = k$ and $U_0(U(C \text{ and } C'))$ holds, where $U = \text{unify}(S_1,S'_1)$.

Proof Since there are no variables used in common by $S_1$ and $S_2$, $U_1 \cup U_2$ is a unifier of $S_1$ and $S'_1$, and $U_1 \cup U_2(C \text{ and } C')$ holds. Hence, there exists a substitution $U'$ such that $U' = \text{unify}(S_1,S'_1) \neq \text{failure}$. Since unification produces a most general unifier, there must exist a ground substitution $U_0$ such that $U_0 \circ U' = U_1 \cup U_2$. $U_0$ is the desired substitution. □

Proposition 3.4 If $a_1^{S_1}$ and $a_2^{S_2}$ are complete, then so is $(a_1*a_2)^{\text{star}(S_1,S_2)}$.

Proof Say that $T_{a_1*a_2}(k) = T_{merge(K_2,K_1')}(T_{a_1}(k),T_{a_2}(k)) \neq \text{ns}$. By the completeness of $S_1$ and $S_2$, there exist ground substitutions $U_1$ and $U_2$ such that $U_1(S_1) = k$ and $U_1(C)$ holds, and $U_2(S'_1) = k$ and $U_2(C')$ holds. By Lemma 3.2, there exists a ground substitution $U_0$ such that $U_0(U(S_1)) = k$ and $U_0(U(C \text{ and } C'))$ holds, where $U = \text{unify}(S_1,S'_1)$. By the completeness of $S_{merge}$, there exists a ground substitution $U'_0$ such that $U'_0 = U'_0 \circ U'$, where $U',S' = S_{merge}(U(S_2),U(S'_2))$. Choose $U'_0$ as the needed substitution, since $U'_0(U'(U(S_1))) = k$ and $U'_0(U'(U(C \text{ and } C')))$. □
Concatenation composition

For action $a^S$, where $a : K \Rightarrow D$ and $K \subseteq D$, we define concatenation composition as $(> a)^{\text{override}(S)}$, where

$$\text{override}(S_1 \Rightarrow S_2 \text{ if } C) = $$

$$U(S_1) \Rightarrow S \text{ if } U(C),$$

if $U, S = S_{\text{concat}}(\text{declarative-facet-scheme-in}(S_1), S_2) \neq \text{failure}$

and $\text{is-satisfiable}(U(C))$

$$\text{ns} \Rightarrow \text{ns} \text{ if false, otherwise}$$

Concatenation composition depends on the type inference defined by $S_{\text{concat}}$. (See Figure 3.6.) $S_{\text{concat}}(d_1, d_2)$ requires that $d_2$ have an "exactly" tag. Problems arise when row and field variables appear in $d_2$. For example, if identifier $A$ is bound to a $\text{present}(\theta_1)$-field in $d_1$, but is bound to a $\Delta_A$-field in $d_2$, we must wait to find out whether $\Delta_A$ will be instantiated to an $\text{absent}$-field or to a $\text{present}$-field to know if it should override the field in $d_1$. To solve this problem, we could put a new field $A : \Delta_1$ in the new target scheme, and use a constraint of the form $(\Delta_1 = \text{present}(\theta_1) \text{ and } \Delta_A = \text{absent}) \text{ or } (\Delta_1 = \text{present}(\theta_1) \text{ and } \Delta_A = \text{present}(\theta_1))$. Fortunately, action semantics expressions have the property that any row and field variables in $d_2$'s typing scheme must also be present in $d_1$'s typing scheme, so the omitted cases in Figure 3.6 never arise. This will be discussed later.

As an example, we show the type inference for $> (\text{find } B; \text{bind } A)$. The type inference for $>$ applies $S_{\text{concat}}$ to the source and target schemes of $\text{find } B; \text{bind } A \in \{ B : \text{present}(\theta_1) \} \alpha_1 \Rightarrow \{ A : \text{present}(\theta_1) \}$ exactly. Both records are extended to have the same fields: the source is extended with an $A : \Delta_1$-field, and the target is extended
\( S_{\text{concat}} : \text{Record} \times \text{Record} \rightarrow \text{Substitution} \times \text{Record} \)
\[
S_{\text{concat}}(d_1, d_2) = \\
\quad \text{let } I = \text{fields-in}(d_1) \cup \text{fields-in}(d_2) \\
\quad \text{let } U_1, d'_1 = \text{extend-record}(I \setminus \text{fields-in}(d_1), d_1) \\
\quad \text{let } U_2, d'_2 = \text{extend-record}(I \setminus \text{fields-in}(d_2), d_2) \\
\quad \text{let } \{\text{fields}_1\}r_1 = d'_1 \\
\quad \text{let } \{\text{fields}_2\}r_2 = d'_2 \\
\quad \text{let } U_3, r = \text{override-row-vars}(r_1, r_2) \\
\quad \text{let } \text{fields} = \text{override-fields}(I, \text{fields}_1, \text{fields}_2) \\
\quad \text{in } U_3 \circ U_2 \circ U_1, \{\text{fields}\}r
\]

\( \text{override-row-vars} : \text{Row} \times \text{Row} \rightarrow \text{Substitution} \times \text{Row} \)
\[
\text{override-row-vars}(r_1, r_2) = \text{cases } (r_1, r_2) \text{ of } \\
\quad (r_1, \text{exactly}) \Rightarrow [\], r_1 \\
\quad (\alpha, \text{exactly}, \beta, \text{exactly}) \Rightarrow [\], \alpha, \text{exactly} \\
\quad (\alpha, \beta) \Rightarrow [\alpha \mapsto \beta, \text{exactly}] \\
\quad (\text{all other cases}) \Rightarrow \text{do not appear in our system}
\]

\( \text{override-fields} : \text{Labels} \times \text{Record} \times \text{Record} \rightarrow \text{Record} \)
\[
\text{override-fields}(\text{nil}, d_1, d_2) = \{\}
\]
\[
\text{override-fields}(i :: \ell, d_1, d_2) = \\
\quad \text{let } d = \text{override-fields}(\ell, d_1, d_2) \\
\quad \text{let } f = \text{cases } (d_1.i, d_2.i) \text{ of } \\
\quad \quad (f, \text{absent}) \Rightarrow f \\
\quad \quad (f, \text{present}(v)) \Rightarrow \text{present}(v) \\
\quad \quad (\Delta, \Delta) \Rightarrow \Delta \\
\quad \quad (\Delta, \Delta') \text{ or } (\text{absent}, \Delta) \text{ or } (\text{present}(v), \Delta) \Rightarrow \text{do not appear in our system}
\]
\[
in \{i : f\}@d
\]

Figure 3.6. Type Inference for Concatenation
with a $B: \text{absent}$-field. This produces the substitution $U = [\alpha_1 \mapsto \{A : \Delta_1\} \alpha_2]$. Next, $\text{override}$-rows is applied to $\alpha_2$ and exactly, with the unknown fields denoted by $\alpha_2$ carried through to the new target scheme. The $\text{present}$-fields in the target override the same fields in the source. Thus, the $A$-field in $\{A : \text{present}(\theta_1), B : \text{absent}\}$ exactly appears in the new target scheme, overriding the $\Delta_1$-field in $\{B : \text{present}(\theta_1), A : \Delta_1\} \alpha_2$. Fields which are "absent" in the target are allowed to flow through from the source. Thus, the $B$-field from the source appears in the new target scheme. $U$ is applied to the source to obtain the action's typing scheme:

$$\triangleright (\text{find } B \ ; \ \text{bind } A) \in \{B : \text{present}(\theta_1), A : \Delta_1\} \alpha_2$$

$$\triangleright \{A : \text{present}(\theta_1), B : \text{present}(\theta_1)\} \alpha_2$$

The scheme illustrates that field $A$ can be either "present" or "absent" in the source of the composite action.

**Soundness and completeness of concatenation composition** The definitions of soundness and completeness for $\text{concat}$ are similar to those for $\text{merge}(D,D)$. (See Propositions 3.1 and 3.2.) We now show soundness and completeness for $\text{override}$.

**Proposition 3.5** If scheme $S = S_1 \rightarrow S_2$ if $C$ is sound for action $a: K \rightarrow D$, where $K \leq D$, then $(\triangleright a) \text{override}(S)$ is sound.

**Proof** Let $\text{override}(S) = S \rightarrow S'$ if $D$, let $k \in K$, and let $U_0$ be a ground substitution such that $U_0(S) = k \neq \text{ns}$ and $U_0(D)$ holds true. Then $U,S' = \text{concat}(\text{declarative-facet-in}(S_1), S_2) \neq \text{failure}$, since $k \neq \text{ns}$. Thus, $k = U_0(U(S_1))$ and $U_0(U(C)) = \text{true}$. By the soundness of $a$, $T_\alpha(k) = U_0(U(S_2)) = d' \neq \text{ns}$. Let $d = \text{declarative-facet-in}(k)$. By the soundness of $\text{concat}$, $T_{\text{concat}}(d, d') = U_0(S')$. Thus, $T_{\triangleright a}(k) = T_{\text{concat}}(d, d') = U_0(S')$. $\square$
Proposition 3.6 If scheme $S = S_1 \dashv S_2$ if $C$ is complete for action $a: K \vdash D$, where $K \leq D$, then $(> a)^{override(S)}$ is complete.

Proof Say that $U, S = S_{concat}(declarative-facet-in(S_1), S_2) \neq failure$, and that $T_{oa}(k) \neq ns$ and $T_a(k) = d' \neq ns$, where $k \in K$. By the completeness of $a^S$, there exists a ground substitution $U_0$ such that $U_0(S_1) = k$ and $U_0(C)$ holds. Let $d = declarative-facet-in(k)$. $T_{concat}(d, d') \neq ns$, since $T_{oa}(k) \neq ns$. By the completeness of $S_{concat}$, there exists a ground substitution $U'_0$ such that $U_0 = U'_0 \circ U$. Choose $U'_0$ to be the needed substitution, since $U_0(S_1) = U'_0(U(S_1)) = k$ and $U_0(C) = U'_0(U(C))$ holds. □

Simplifying properties

The definitions of $S_{merge}$ and $S_{concat}$ in Figures 3.5 and 3.6 are well-defined, despite the omitted cases. Action semantics expressions possess two important simplifying properties that allow their omission:

**Alpha-preserved** A scheme $S_1 \dashv S_2$ if $C$ with kinding $K_1 \vdash K_2$ has the $\alpha$-preserved property if whenever $K_2 \leq D$ and the declarative facet scheme in $S_2$ contains a row variable $\alpha$, then $K_1 \leq D$ and the declarative facet scheme in $S_1$ contains the same $\alpha$.

**Delta-preserved** A scheme $S_1 \dashv S_2$ if $C$ with kinding $K_1 \vdash K_2$ has the $\Delta$-preserved property if whenever $K_2 \leq D$ and the declarative facet scheme in $S_2$ contains a field $I: \Delta$, then $K_1 \leq D$ and the declarative facet scheme in $S_1$ contains the same field $I: \Delta$. 
Proposition 3.7 Let \( a_1 \) have scheme \( S_1 = S_1' \vdash S_2 \) if \( C_1 \), and let \( a_2 \) have scheme \( S_2 = S_2' \vdash S'_2 \) if \( C_2 \). If \( S_1 \) and \( S_2 \) have the \( \alpha \)-preserved and \( \Delta \)-preserved properties for all row variables \( \alpha \), and all field variables \( \Delta \), then:

(ii) if \( U = \text{unify}(S_1, S'_1) \neq \text{failure} \), then \( U(S_1) \) and \( U(S'_2) \) have the \( \alpha \)- and \( \Delta \)-preserved properties, and

(i) if \( U = \text{unify}(S_2, S'_2) \neq \text{failure} \), then \( U(S_1) \) and \( U(S'_2) \) have the \( \alpha \)- and \( \Delta \)-preserved properties.

Proposition 3.7 is needed to show that the \( \alpha \)- and \( \Delta \)-preserved properties of an action's typing scheme are preserved by the type inference algorithms for the combinators. Informally, consider the type inference for \( a_1 \times a_2 \), where \( d_1 \vdash d_2 \) if \( C \) is the scheme for \( a_1: D \rightarrow D \), and \( d'_1 \vdash d'_2 \) if \( C' \) is the scheme for \( a_2: D \rightarrow D \). Assume that the schemes have the \( \alpha \)- and \( \Delta \)-preserved properties. The type inference unifies \( d_1 \) with \( d'_1 \), producing the substitution \( U \), and forcing the row and field variables to be the same in \( U(d_1) \) and \( U(d'_1) \). If \( U(d_2) \) has a row variable, by Proposition 3.7 (i), it is in \( U(d_1) \) as well, and similarly for \( U(d'_1) \). So if both \( U(d_2) \) and \( U(d'_2) \) have row variables, they must be identical, implying that the two arguments to \( S_{\text{union}} \) have the same row variables. Thus, the omitted cases in Figure 3.5 never arise. A similar result holds for field variables in \( U(d_1) \) and \( U(d'_2) \).

Proposition 3.7 (ii) is used in a similar manner to show that the type inference for \( a_1 \; ; \; a_2 \) preserves the properties.

For type inference on \( \triangleright \; a \), if \( a \)'s scheme \( d_1 \vdash d_2 \) if \( C \) has the \( \alpha \)- and \( \Delta \)-preserved properties, then the row and field variables in \( d_2 \) also appear in in \( d_1 \). Again, the omitted cases in Figure 3.6 never arise.
Conditional composition

Conditional composition requires modification of the syntax for action typing schemes:

\[ S ::= S_1 \rightarrow S_2 \text{ if } C \mid S_1/S_2 \]

The symbol / may be thought of as an “exclusive or”. The scheme for the action \( \text{succ}/\text{not} \) looks like \( \theta \rightarrow \theta \text{ if } \theta \leq \text{real} / \text{bool} \rightarrow \text{bool} \). Note that / is not a type constructor. (Recall that the typing function for \( \text{succ}/\text{not} \) is defined as \( T_{\text{succ/not}}(t) = \left\{ \begin{array}{ll} S & \text{if } T_{\text{succ}}(t) \neq \text{ns} \text{ then } T_{\text{succ}}(t) \text{ else } T_{\text{not}}(t). \end{array} \right\} \))

Because there are two forms for action schemes now, we can no longer talk about the “source” and “target” of an action. We remedy this by introducing two new constructs into our syntax for typing schemes:

\[ sc \in \text{Scheme-constraint} \]
\[ sl \in \text{Scheme-list} \]

\[ sc ::= S, C \mid sc_1/sc_2 \]
\[ sl ::= S \mid sl_1/sl_2 \]

We use the following functions to extract the source and target of a scheme:

\[ \text{source: Typing-scheme } \rightarrow \text{ Scheme-constraint} \]
\[ \text{source}(S_1 \rightarrow S_2 \text{ if } C) = S_1, C \]
\[ \text{source}(S_1/S_2) = \text{source}(S_1)/\text{source}(S_2) \]

\[ \text{target: Typing-scheme } \rightarrow \text{ Scheme-constraint} \]
\[ \text{target}(S_1 \rightarrow S_2 \text{ if } C) = S_2, C \]
\[ \text{target}(S_1/S_2) = \text{target}(S_1)/\text{target}(S_2) \]
Each source and target of the slash scheme must be kept together with its associated constraint set because the application of a substitution \( U \) to a source or target scheme \( S \) depends on the constraint set \( U(C) \) being satisfiable. Substitution into a Scheme-constraint yields a Scheme-list:

\[
(\cdot) : \text{Substitution} \times \text{Scheme-constraint} \rightarrow \text{Scheme-list}
\]

\[
\text{failure}(sc) = \text{ns}
\]

\[
U(sc) = \text{cases } sc \text{ of}
\]

\[
\begin{align*}
& S, C \Rightarrow \text{if is-satisfiable}(UC) \text{ then } U(S) \text{ else } \text{ns} \\
& sc_1/sc_2 = U(sc_1)/U(sc_2)
\end{align*}
\]

where \( \text{ns}/S = S/\text{ns} = S \). A consequence of the above definition is that if \( U(sc_1/sc_2) = sl_1/sl_2 \), where \( sl_1 \neq \text{ns} \) and \( sl_2 \neq \text{ns} \), then \( sl_1/sl_2 \neq S \) for any scheme \( S \).

The definition of the typing function for slash depends on the domains of the typing functions for the two actions being disjoint. The type inference for slash must determine whether this holds by examining the source schemes of the two actions. The function \( \text{indep} \) takes care of this:

\[
\text{indep}: \text{Scheme-constraint} \times \text{Scheme-constraint} \rightarrow \text{Bool}
\]

\[
\text{indep}((S_1,C_1),(S_2,C_2)) =
\]

\[
\begin{align*}
\text{false}, & \quad \text{if } U = \text{unify}(S_1,S_2) \neq \text{failure} \text{ and is-satisfiable}(UC_1), \\
& \hspace{2cm} \text{and is-satisfiable}(UC_2) \\
\text{true}, & \quad \text{otherwise}
\end{align*}
\]

\[
\text{indep}(S,C), sc_1/sc_2) = \text{indep}((S,C),sc_1) \text{ and } \text{indep}((S,C),sc_2)
\]

\[
\text{indep}(sc_1/sc_2,sc) = \text{indep}(sc_1,sc) \text{ and } \text{indep}(sc_2,sc)
\]
Proposition 3.8  \( \text{indep}(s_{c_1}, s_{c_2}) \) implies there does not exist a substitution \( U \) such that \( U(\text{source } s_{c_1}) = U(\text{source } s_{c_2}) \neq \text{ns} \).

We now define the type inference for conditional composition and show its soundness and completeness. The incorporation of this new form of scheme into the type inference for the other combinators and their proofs of soundness and completeness can be found in [8].

For actions \( a_1^{S_1} \) and \( a_2^{S_2} \), where \( a_1 \) and \( a_2 \) both have kinding \( K_1 \rightarrow K_2 \), we define the type inference for conditional composition as \( (a_1/a_2)^{\text{slash}(S_1,S_2)} \), where

\[
\text{slash}(S_1,S_2) =
\begin{align*}
S_1/S_2, & \quad \text{if indep(source}_S_1,\text{source}_S_2) \\
\text{ns} \not\rightarrow \text{ns if false}, & \quad \text{otherwise}
\end{align*}
\]

Recall that the typing function for conditional composition is

\[
T_{a_1/a_2} = T_{\text{fail}(K_2)} \circ T_{a_1}, \quad \text{when domain}(T_{a_1}) \cap \text{domain}(T_{a_2}) \neq \emptyset
\]

\[
\lambda k \in K_1. \text{ if } T_{a_1}(k) \neq \text{ns then } T_{a_1}(k) \text{ else } T_{a_2}(k), \quad \text{otherwise}
\]

Proposition 3.9 If \( a_1^{S_1} \) and \( a_2^{S_2} \) are complete, then so is \( (a_1/a_2)^{\text{slash}(S_1,S_2)} \).

Proof Let \( a_1 \) and \( a_2 \) have kinding \( K_1 \rightarrow K_2 \), and let \( k \in K_1 \). Say that \( T_{a_1/a_2}(k) \neq \text{ns} \). Without loss of generality, say that \( T_{a_1/a_2}(k) = T_{a_1}(k) \neq \text{ns} \), and \( T_{a_1}(k) = \text{ns} \). By the completeness of \( S_1 \), there exists a ground substitution \( U_1 \) such that \( U_1(\text{source}_S_1) = k \). Suppose there is also some ground substitution \( U_2 \) such that \( U_2(\text{source}_S_2) = k \). Since there are no common variables, \( U_1 \cup U_2 \) is a unifier of \( \text{source}_S_1 \) and \( \text{source}_S_2 \), contradicting Proposition 3.8. So choose any substitution \( U' \) that is ground for \( S_2 \): \( U_1 \cup U'_2(\text{source } \text{slash}(S_1,S_2)) = k \). \( \square \)
Lemma 3.3 If $a_1^{S_1}$ and $a_2^{S_2}$ are complete, then indep($source_{S_1}, source_{S_2}$) implies $domain(T_{a_1}) \cap domain(T_{a_2}) = \emptyset$.

Proof Assume indep($source_{S_1}, source_{S_2}$) and suppose there is some type $k \in domain(T_{a_1}) \cap domain(T_{a_2})$. By completeness, there exist ground substitutions $U_1$ and $U_2$ such that $U_1(source_{S_1}) = k$ and $U_2(source_{S_2}) = k$. Since $S_1$ and $S_2$ share no variables in common, $U_1 \cup U_2$ is a unifier of $source_{S_1}$ and $source_{S_2}$, contradicting Proposition 3.8. □

Proposition 3.10 If $a_1^{S_1}$ and $a_2^{S_2}$ are sound and complete, then $(a_1/a_2)^{slash(S_1,S_2)}$ is sound.

Proof Let $U$ be a ground substitution such that $U(source \slash (S_1,S_2)) = k \neq ns$, for some type $k \in K_1$. There are two cases:

(i) $T_{a_1/a_2} = T_{fail(K_2)} \circ T_{a_1}$: Then $domain(T_{a_1}) \cap domain(T_{a_2})$ is non-empty. By Lemma 3.3, indep($source_{S_1}, source_{S_2}$) does not hold, hence $\slash(S_1,S_2)$ has scheme ns $\Rightarrow$ ns if false, and the result vacuously holds.

(ii) $T_{a_1/a_2} = \lambda k \in K_1$, if $T_{a_1}(k) \neq ns$ then $T_{a_1}(k)$ else $T_{a_2}(k)$: Then $domain(T_{a_1}) \cap domain(T_{a_2}) = \emptyset$. Without loss of generality, assume that $T_{a_1}(k) \neq ns$ and that $T_{a_2}(k) = ns$. If $\slash(S_1,S_2) = ns \Rightarrow$ ns if false, we are finished. If it is $S_1/S_2$, say that $U(source_{S_1}/S_2) = k \neq ns$. By the definition of substitution, $U(source_{S_1}) = k$ xor $U(source_{S_2}) = k$. If $U(source_{S_1}) = k$, by the soundness of $S_1$, we are finished. If $U(source_{S_2}) = k$, by the soundness of $S_2$, $T_{a_2}(k) = U(target_{S_2}) = ns$. □
CHAPTER 4. THE SATISFACTION OF CONSTRAINTS

The type inference algorithms presented in the previous chapter make use of an algorithm \textit{is-satisfiable} which tests whether a set of constraints is satisfiable, that is, whether there exists a ground substitution which makes all the constraints hold true. In this chapter we describe the algorithms for checking satisfiability. Our algorithms are an extension of those introduced by Mitchell [20] and implemented by Fuh and Mishra [10, 11] to record types with inclusion polymorphism.

Type Inference With Subtypes

In [20], Mitchell extends ML-style type inference with subtypes. Subtyping is interpreted as subset inclusion ($\sigma \leq \tau$ implies $\llbracket \sigma \rrbracket \subseteq \llbracket \tau \rrbracket$), and a minimal set of atomic coercions (i.e., what we call "constraints") is used in the typing schemes. An atomic coercion is of the form $\tau \leq \tau'$, where $\tau$ and $\tau'$ are either primitive types or type variables. Non-atomic coercions are derived as logical consequences from this set of atomic coercions by rules such as:

\( \text{(arrow)} \) from $\sigma' \leq \sigma$ and $\tau \leq \tau'$, derive $\sigma \rightarrow \tau \leq \sigma' \rightarrow \tau'$

The type inference algorithm assigns types to terms in the lambda calculus. Subtyping is introduced by the coercion rule:

\( \text{(coerce)} \) if $e$ has type $\sigma$ and $\sigma \leq \tau$, then $e$ has type $\tau$
which "inserts" coercions into the type of a function application. Mitchell uses an algorithm called COERCE to determine for a coercion the minimal atomic coercion set that implies it.

Legal typings of an expression are obtained by substitution into its type scheme. The substitution must "respect" the scheme's coercion set: a substitution $U$ respects coercion set $C$ if for every coercion $\sigma \leq \tau$ in $C$, $U(\sigma)$ matches $U(\tau)$. Two expressions "match" if they have the same form: type expressions $\sigma$ and $\tau$ match if and only if both are atomic, or both are applications of the same type constructor (i.e., $\sigma = \sigma_1 \rightarrow \sigma_2$, and $\tau = \tau_1 \rightarrow \tau_2$) and their corresponding subexpressions match (i.e., $\sigma_1$ matches $\tau_1$, and $\sigma_2$ matches $\tau_2$). The algorithm MATCH computes this equivalence relation on the structure of types.

Maintaining the set of atomic constraints depends on the algorithms COERCE and MATCH, however, their definitions were omitted from [20].

Fuh and Mishra's Satisfaction Algorithm

Fuh and Mishra implemented algorithms to check the satisfiability of a set of constraints based on Mitchell's work [10]. The algorithm MATCH is applied to an initial set of constraints $C_0$ to obtain a "minimal matching substitution" $U_M$ which instantiates the type variables in $C_0$ with structure necessary for $C_0$ to be satisfied. For example, given the constraint $\theta \leq \text{int} \times \text{real}$, we know from the subtype ordering that $\theta$ must have at least as much structure as a product type. Any valid instance of the constraint must instantiate $\theta$ to a product. Thus, the substitution returned by MATCH would instantiate $\theta$ to a product type, i.e., $U_M = [\theta \mapsto \theta_1 \times \theta_2]$. Valid instances may be obtained by further instantiations of $\theta_1$ and $\theta_2$. MATCH fails when
the type constructors of the two terms in a coercion differ (e.g., \( \theta_1 \times \theta_2 \leq \theta_3 \rightarrow \theta_4 \)).

An algorithm called SIMPLIFY iterates over the constraint set \( U_M(C_0) \), replacing each coercion by the set of coercions with less structure which entail it, until all coercions that remain are between atomic type schemes. For example, the constraint \( \theta_1 \times \theta_2 \leq \text{int} \times \text{real} \) would be replaced by the set of coercions \( \{ \theta_1 \leq \text{int}, \theta_2 \leq \text{real} \} \).

Finally, an algorithm called CONSISTENT is applied to the set of atomic constraints. CONSISTENT tests whether there exists a ground substitution which, when applied to the set of constraints, makes all of the constraints hold true. CONSISTENT is described in the next section.

Consistency of atomic constraints

Atomic constraints have the following forms: \( p \leq p' \), \( p \leq \theta \), \( \theta \leq p \), and \( \theta \leq \theta' \). The first form can be easily checked and eliminated from the constraint set. We will assume this has been done and describe Fuh and Mishra's algorithm for the remaining forms.

Let \( C_A \) be a set of atomic constraints. For all type variables \( \theta \) that occur in \( C_A \), let \( I_\theta \) stand for the set of types that \( \theta \) can be instantiated to. \( I_\theta \) is initialized to the set of primitive types to indicate that there are no constraints on \( \theta \). For our system, \( \text{Prim} = \{ \text{bool}, \text{nat}, \text{int}, \text{real} \} \). The following operations are used: for \( p \in \text{Prim} \), define \( \uparrow p = \{ p' \in \text{Prim} \mid p \leq p' \} \), and \( \downarrow p = \{ p' \in \text{Prim} \mid p' \leq p \} \); and for \( I_\theta \), define \( \uparrow I_\theta = \bigcup_{p \in I_\theta} \uparrow p \), and \( \downarrow I_\theta = \bigcup_{p \in I_\theta} \downarrow p \).

Algorithm CONSISTENT is specified in Figure 4.1. It iterates over the set of equations \( I_\theta \) until some \( I_\theta \) converges to \( \emptyset \), or all \( I_\theta \) remain unchanged. Convergence to \( \emptyset \) indicates that there does not exist an instantiation for \( \theta \) which makes the
for all $e \in C_A$ do
  cases $e$ of
    $p \leq \theta \Rightarrow$
    \begin{align*}
    I'_\theta & \leftarrow \downarrow \bigcup I_\theta; \\
    \text{if } I'_\theta = \emptyset \text{ then failure } \text{ else } stable & \leftarrow \text{stable } \land I_\theta = I'_\theta
    \end{align*}
  \quad \text{if } I'_\theta = \emptyset \text{ then failure } \text{ else } stable \leftarrow \text{stable } \land I_\theta = I'_\theta
  \quad \begin{align*}
    \theta \leq p \Rightarrow
    I'_\theta & \leftarrow \downarrow \bigcup I_\theta; \\
    \text{if } I'_\theta = \emptyset \text{ then failure } \text{ else } stable & \leftarrow \text{stable } \land I_\theta = I'_\theta
  \end{align*}
  \quad \begin{align*}
    \theta \leq \theta' \Rightarrow
    I'_\theta & \leftarrow \downarrow \bigcup I_\theta; \\
    I'_\theta & \leftarrow \downarrow \bigcup I_\theta; \\
    \text{if } I'_\theta = \emptyset \text{ or } I'_\theta = \emptyset \text{ then failure } \\
    \text{else stable } \leftarrow \text{stable } \land I_\theta = I'_\theta \land I'_\theta = I'_\theta
  \end{align*}
until stable

Figure 4.1. Fuh and Mishra's algorithm \textsc{consistent}

constraints hold true: the constraint set is inconsistent.

Fuh and Mishra state a property that the equations satisfy when the algorithm terminates successfully and conjecture that it guarantees the consistency of $C_A$, but they do not prove it.

Note that \textsc{consistent} does not return a substitution, it merely tells us whether one exists. For example, given the constraint set $C_A = \{\theta_1 \leq \text{int}, \theta_2 \leq \theta_1\}$, the algorithm terminates with the answer \textit{true}, and the equations: $I_{\theta_1} = \{\text{nat, int}\}$ and $I_{\theta_2} = \{\text{nat, int}\}$. The equations do not give us a solution, however, as we do not know which type to choose for each type variable. For example, if we choose the substitution $U_A = [\theta_1 \mapsto \text{nat}][\theta_2 \mapsto \text{int}]$, we find $U_0(C_A)$ is false. We could use a brute force approach to find a solution, but this is impractical when the constraint sets are large. Luckily, the type inference algorithms presented in the previous chapter only need to know whether such a substitution exists.
Extension to Record and Storage Type Constraints

We have extended Fuh and Mishra's satisfaction algorithms to operate upon record and storage type constraints. In the presence of inclusion subtyping on records, our types no longer possess their notion of structural equivalence called “matching”. We have therefore renamed our extended version of Fuh and Mishra's MATCH algorithm decompose.

Another difference in the general structure of our algorithms is that we apply simplify at intermediate steps during decomposition, as is done in [20]. We do this mainly for ease of implementation: as records tend to be large, by applying simplify to a record constraint at intermediate steps, we can ignore fields of the record which have already been decomposed, thus making further simplification steps on the record easier.

Distinguished constraint forms

In addition to atomic constraint forms, algorithm decompose distinguishes the following constraint forms, which we call M-forms:

\[ \{i : f_i, j : f_j\}r \leq \{i : \Delta_i\}\beta \]
\[ \{i : \Delta_i\}a_{\text{exactly}} \leq \{i : \Delta_i'\}b_{\text{exactly}} \]
\[ \{i : \Delta_i\}a_{\text{exactly}} \leq \{i : \Delta_i'\}a_{\text{exactly}} \]
\[ s \leq (\lambda)\sigma \]

These constraint forms are distinguished because they have multiple solutions; we cannot further instantiate any of their type variables without more information, i.e., without other constraints involving the variables. The first three forms are record
constraints for which we do not have enough information to fill in types for the $\Delta$-variables. For example, the instantiation of $\Delta_I$ in record $\{I : \Delta_I\text{exactly} \leq \{I : \Delta'_I\text{exactly}\}$ would depend on whether $\Delta'_I$ were instantiated to an absent-field or a present-field. The last form is a storage type constraint for which further instantiation of $\sigma$ will determine the instantiation of $s$. Any cells that $\langle \rangle \sigma$ is extended with should be added to $s$.

Decomposition

Algorithm decompose is specified in Figure 4.2. decompose iterates over a constraint set. At iteration $i$, it chooses a constraint $e_i$ from $C_{i-1}$ whose form is not distinguished. The function decompose-constraint is applied to $e_i$, giving a substitution $U_{e_i}$ that instantiates type variables in $e_i$ to terms with more structure. $U_{e_i}$ corresponds to the substitution produced by Fuh and Mishra's MATCH algorithm when applied to a single constraint. Next, function simplify is applied to $U_{e_i}(e_i)$ to produce a simplified set of constraints $C_{e_i}$ with less structure and fewer fields. (See Figure 4.3.) Substitution $U_{e_i}$ is applied to the remaining constraints $C_{i-1} - e_i$, which are then unioned with the the simplified set of constraints $C_{e_i}$ that replace $e_i$ for the next iteration. The substitutions generated at each iteration are accumulated. decompose terminates when all constraints have M-forms.

The function decompose-constraint uses the functions decompose-record-constraint and decompose-storage-constraint which are described in a later section. First, we discuss simplify.
decompose : Substitution × Constraints → Substitution × Constraints

decompose(U_{i-1}, C_{i-1}) = U_i, C_i

where \( U_i = U_{i-1} \) and \( C_i = C_{i-1} \), if all constraints in \( C_{i-1} \) have M-forms

otherwise,

\[
\begin{align*}
U_i &= U_{e_i} \circ U_{i-1} \\
C_i &= C_{e_i} \cup U_{e_i}(C_{i-1} - \{e_i\}) \\
U_{e_i} &= \text{decompose-constraint}(e_i) \\
C_{e_i} &= \text{simplify}(U_{e_i}(e_i))
\end{align*}
\]

and \( e_i \) is a constraint in \( C_{i-1} \) that does not have an M-form

decompose-constraint : Constraint → Substitution

decompose-constraint(e) = cases e of

\[
\begin{align*}
v_1 \times v_2 &\leq v_1' \times v_2' \text{ or } v_1 \rightarrow v_2 \leq v_1' \rightarrow v_2' \Rightarrow [] \\
\theta &\leq v_1 \times v_2 \text{ or } v_1 \times v_2 \leq \theta \Rightarrow \\
&\text{if occurs-in}(\theta, v_1 \times v_2) \text{ then failure else } [\theta \mapsto \theta_1 \times \theta_2] \\
\theta &\leq v_1 \rightarrow v_2 \text{ or } v_1 \rightarrow v_2 \leq \theta \Rightarrow \\
&\text{if occurs-in}(\theta, v_1 \rightarrow v_2) \text{ then failure else } [\theta \mapsto \theta_1 \rightarrow \theta_2] \\
\langle l : v' \rangle\text{var} &\leq v \text{ or } v \leq \langle l : v' \rangle\text{var} \Rightarrow \text{unify}(\langle l : v' \rangle\text{var}, v) \\
s &\leq s' \Rightarrow \text{decompose-storage-constraint}(s \leq s') \\
d &\leq d' \Rightarrow \text{decompose-record-constraint}(d \leq d') \\
\theta &\leq d \Rightarrow \text{let } U_1 = [\theta \mapsto \text{\{\}}\alpha] \\
&\text{let } U_2 = \text{decompose-record-constraint}(\text{\{\}}\alpha \leq U_1(d)) \\
&\text{in } U_2 \circ U_1 \\
d &\leq \theta \Rightarrow \text{let } U_1 = [\theta \mapsto \text{\{\}}\alpha] \\
&\text{let } U_2 = \text{decompose-record-constraint}(U_1(d) \leq \text{\{\}}\alpha) \\
&\text{in } U_2 \circ U_1 \\
\text{otherwise} &\Rightarrow \text{failure}
\end{align*}
\]

Figure 4.2. Algorithms decompose and decompose-constraint
Algorithm \textit{simplify} maps a "matching" constraint $e$ into a set of simplified constraints which entail it. Simplification of a storage constraint removes those storage cells which appear in both schemes. For example, the constraint $(\ell_2: \text{int}, \ell_3: \text{bool}) \sigma_1 \leq (\ell_2: \text{int}) \sigma_2$ simplifies to $(\ell_3: \text{bool}) \sigma_1 \leq (\ell_3: \text{bool}) \sigma_2$. Simplification of the record constraint \{A: absent, B: present($v$), C: $\Delta$\}$\alpha_{\text{exactly}} \leq \{A: absent, B: present($v'$), C: $\Delta'$\}$\beta_{\text{exactly}} eliminates the relation on the A-absent fields and moves the relation on the B-present fields to the constraint $v \leq v'$ on the functional facet, leaving the constraint \{C: $\Delta$\}$\alpha_{\text{exactly}} \leq \{C: $\Delta'$\}$\beta_{\text{exactly}} on the "unknown" field. Simplification of the non-exactly record constraint \{B: present($v$), C: $\Delta$, D: present($v''$)\}$\alpha_{\text{exactly}} \leq \{B: present($v'$), C: $\Delta'$\}$\beta moves the relation on B-present fields to the constraint
Notice that among the constraints resulting from \( \text{simplify}(e) \), those on storage and records all have M-forms. These constraints will thus not be decomposed further until the decomposition on other constraints fills in their row, field, or storage variables.

The set of constraints \( \text{simplify}(e) \) is equivalent to the constraint \( e \) in the sense that any ground substitution which satisfies \( e \) also satisfies \( \text{simplify}(e) \). This is proved as Proposition 4.1. The following fact allows us to consider subsets of the fields in a record constraint as long as we remember whether the fields were related by an exactly ordering or a non-exactly ordering.

**Fact 4.1** For all ground types and ground type schemes, where \( I \) and \( J \) are disjoint sets of labels, and \( r \) is either \( \{ \} \) or \( \{ \}^{\text{exactly}} \):

(a) \( \{ i: f_i, j: f_j \}^{\text{exactly}} \leq \{ i: f'_i, j: f'_j \}^{\text{exactly}} \) if and only if

\( \{ i: f_i \}^{\text{exactly}} \leq \{ i: f'_i \}^{\text{exactly}} \) and \( \{ j: f_j \}^{\text{exactly}} \leq \{ j: f'_j \}^{\text{exactly}} \)

(b) \( \{ i: f_i, j: f_j \}_r \leq \{ i: f'_i, j: f'_j \} \) if and only if

\( \{ i: f_i \}_r \leq \{ i: f'_i \} \) and \( \{ j: f_j \}_r \leq \{ j: f'_j \} \)

(c) \( \{ i: f_i, j: f_j \}_r \leq \{ i: f'_i \} \) if and only if

\( \{ i: f_i \}_r \leq \{ i: f'_i \} \) and \( \{ j: f_j \}_r \leq \{ \} \)

We make a critical assumption in algorithm \( \text{simplify} \) that for all row variables \( \alpha \), we can always determine \( \alpha \)'s domain of use regardless of its context, i.e., the index set with which it appears. This assumption is needed because our algorithm removes decomposed fields from a record, yet leaves the record's row variable paired with the remaining \( \Delta \)-fields. For example, the constraint \( \{ B: \text{present}(v), C: \Delta \}^{\text{exactly}} \leq \)}
\{B: \text{present}(v'), C: \Delta'\}B_{\text{exactly}} \text{simplifies to } \{C: \Delta\}a_{\text{exactly}} \leq \{C: \Delta'\}B_{\text{exactly}} \text{ and } v \leq v'. \text{ Identifier } B \text{ no longer appears in the index set of either } \{C: \Delta\}a_{\text{exactly}} \text{ or } \{C: \Delta'\}B_{\text{exactly}} \text{ and we need take care not to extend } \alpha \text{ or } \beta \text{ with a } B\text{-field.}

Knowledge of a row variable's domain of use is needed by the function \textit{extend-record} in Figure 4.5. We will therefore assume that a row variable is paired with it's domain of use, although we will not add an additional data structure for this.

**Proposition 4.1** For all substitutions \(U_0\) that are ground with respect to \(e\), \(U_0(e)\) holds if and only if \(U_0(\text{simplify } e)\) holds.

**Proof** Let \(U_0\) be a ground substitution. The proof is by cases on the form of \(e\):

(i) \(v_1 \times v_2 \leq v'_1 \times v'_2\) : Then, \(U_0(e) = U_0(v_1) \times U_0(v_2) \leq U_0(v'_1) \times U_0(v'_2)\).

By definition of \(\leq\) on product, the right-hand side holds if and only if \(U_0(v_1) \leq U_0(v'_1)\) and \(U_0(v_2) \leq U_0(v'_2)\), and this equals \(U_0(\text{simplify } e)\).

(ii) \(v_1 \rightarrow v_2 \leq v'_1 \rightarrow v'_2\) : This case is similar to the previous.

(iii) \(\langle l : v\rangle \text{var} \leq \langle l : v\rangle \text{var} : \) Any substitution that is ground with respect to \(e\) will satisfy both \(e\) and \(\text{simplify}(e) = \{\}\).

(iv) \(\langle i : m_i, j : m_j\rangle \sigma_1 \leq \langle i : m_i\rangle \sigma_2 : \) Then, \(U_0(e) = \langle i : U_0(m_i), j : U_0(m_j)\rangle U_0(\sigma_1) \leq \langle i : U_0(m_i)\rangle U_0(\sigma_2)\). By the equivalence rewrite rules for ground types and the definition of subtyping on storage types, the right-hand side holds if and only if \(\langle j : U_0(m_j)\rangle U_0(\sigma_1) \leq \langle j \rangle U_0(\sigma_2)\), and this equals \(U_0(\text{simplify } e)\).

(v) \(\{i : \Delta_i, j : v_j, k : \text{absent}\}_{\text{exactly}} \leq \{i : \Delta'_i, j : v'_j, k : \text{absent}\}_{\text{exactly}} : \) Similar to the following case.
(vi) \(\{i : \Delta_i, j : v_j, k : \text{absent}\}_\alpha \leq \{i : \Delta'_i, j : v'_j, k : \text{absent}\}_\beta\).

Then \(U_0(e) = \{i : U_0(\Delta_i), j : U_0(v_j), k : \text{absent}\}_\alpha U_0(\beta) \leq \{i : U_0(\Delta'_i), j : U_0(v'_j), k : \text{absent}\}_\beta\).

(For lack of space, we abbreviate \(\text{present}(v)\) as \(v\).)

Assume \(U_0(\alpha) = \{\ell : f_1\}\), and \(U_0(\beta) = \{\ell : f'_1\}\), for identifier set \(L\), and ground types \(f, f'\). By the equivalence rewrite rules, this holds if and only if \(\{i : U_0(\Delta_i), j : U_0(v_j), \ell : f\}_\alpha \leq \{i : U_0(\Delta'_i), j : U_0(v'_j), \ell : f'_1\}_\beta\).

By Fact 4.1, this holds if and only if \(\{i : U_0(\Delta_i), j : U_0(v_j), \ell : f\}_\alpha \leq \{i : U_0(\Delta'_i), j : U_0(v'_j)\}, \) which equals \(\{i : U_0(\Delta_i)\}_\alpha \leq \{i : U_0(\Delta'_i)\}_\beta\) and \((\text{and}_{j \in J}\{U_0(v_j) \leq U_0(v'_j)\})\), which equals \(U_0(\text{simplify } e)\).

(vii) \(\{j : v_j, \ell : \text{absent}, k : f_k, i : f_i\}_r \leq \{j : v'_j, \ell : \text{absent}, k : \Delta_k\}_\beta\). Then,

\(U_0(e) = \{j : U_0(v_j), \ell : \text{absent}, k : U_0(f_k), i : U_0(f_i)\}_r \leq \{j : U_0(v'_j), \ell : \text{absent}, k : U_0(\Delta_k)\}_\beta\).

By the equivalence rewrite rules and Fact 4.1, this holds if and only if \(\{k : U_0(f_k), i : U_0(f_i)\}_r \leq \{k : U_0(\Delta_k)\}_\beta\) and \((\text{and}_{j \in J}\{U_0(v_j) \leq U_0(v'_j)\})\), which is equal to \(U_0(\text{simplify } e)\). □

Decomposition of storage

Function \(\text{decompose-storage-constraint}\) is defined in Figure 4.4. Decomposition of the storage constraint \((\ell_2 : \theta, \ell_3 : \text{int})\sigma_1 \leq (\ell_2 : \text{real}, \ell_3 : \text{bool})\sigma_2\) produces the substitution \(U_2 \circ U_1\), where \(U_1 = [\sigma_1 \leftarrow (\ell_3 : \text{bool})\sigma_3]\) and \(U_2 = [\theta \leftarrow \text{real}]\). We prove the following lemma about the substitution produced by \(\text{decompose-storage-constraint}\).

Lemma 4.1 Let \(U_e = \text{decompose-storage-constraint}(s \leq s')\), and let \(U_0\) be a ground substitution such that \(U_0(s \leq s')\) is true. Then there exists a ground substitution \(U'\) such that \(U_0 = U' \circ U_e\).
decompose-storage-constraint : Storage-type × Storage-type → Substitution

\[
\text{decompose-storage-constraint}(\langle i : m_i, k : m_k \rangle \sigma_1 \leq \langle i : m_i', j : m_j \rangle \sigma_2) = \\
\text{let } U_1 = [\sigma_1 \mapsto \langle j : m_j \rangle \sigma] \\
\text{let } U_2 = \text{unify-cells}(I, U_1(\langle i : m_i \rangle), U_1(\langle i : m_i' \rangle)) \\
\text{in } U_2 \circ U_1
\]

Figure 4.4. Decomposition of Storage

**Proof** Function \(\text{decompose-storage-constraint}\) extends \(s\) with those cells of \(s'\) which it does not have to produce substitution \(U_1\). By the definition of subtyping on storage, substitution \(U_0\) must also extend \(s\) with these cells. Since unification produces a most general unifier, there exists a ground substitution \(U''\) such that \(U_0 = U'' \circ U_1\).

Next, the cells of \(s'\) are unified with those of \(s\) to produce \(U_2\). Since unification produces a most general unifier, there must exist a ground substitution \(U''\) such that \(U' = U'' \circ U_2\). So \(U_0 = U'' \circ U_2 \circ U_1 = U'' \circ U_0\).

**Decomposition of records**

Function \(\text{decompose-record-constraint}\) is defined in Figure 4.5. Decomposition of the record constraint \(\{B: \text{present}(v), C: \Delta\} \alpha \leq \{A: \text{absent}\} \beta\) exactly extends both records with additional fields to produce substitutions \(U_1 = [\alpha \mapsto \{A: \Delta_2\} \alpha_2]\) and \(U_2 = [\beta \mapsto \{B: \Delta_3, C: \Delta_4\} \beta_2]\). Function \(\text{decompose-rows}\) is applied to \(\alpha_2 \leq \beta_2\) exactly to produce substitution \(U_3 = [\alpha_2 \mapsto \alpha_{3\text{exactly}}]\). Function \(\text{decompose-fields}\) is then applied to \(\{A: \Delta_2, B: \text{present}(v), C: \Delta\}\) and \(\{A: \text{absent}, B: \Delta_3, C: \Delta_4\}\) to produce substitution \(U_4 = [\Delta_2 \mapsto \text{absent}] [\Delta_3 \mapsto \text{present(\theta)}]\). Application of the result substitution \(U_4 \circ U_3 \circ U_2 \circ U_1\) to the original constraint produces the constraint
extend-record : Labels × Record → Substitution × Record
extend-record(J, d) = cases d of
        {i : f}exactly ⇒ [], {i : f, j : absent}exactly
        {i : f}α = [α → {j : Δj}β], {i : f, j : Δj}β
        {i : f}αexactly ⇒ [α → {j : Δj}β], {i : f, j : Δj}βexactly

decompose-rows : Row × Row → Substitution
decompose-rows(r₁ ≤ r₂) = cases r₁ ≤ r₂ of
        α ≤ exactly ⇒ [α → exactly]
        α ≤ βexactly ⇒ [α → α'exactly]
        exactly ≤ βexactly ⇒ [β → { }]
        αexactly ≤ exactly ⇒ [α → { }]
        all other cases ⇒ []

decompose-record-constraint : Record × Record → Substitution
decompose-record-constraint(d₁ ≤ d₂) =
        let I = fields-of(d₁) ∪ fields-of(d₂)
        let U₁, d'₁ = extend-record(I \ fields-of(d₁), d₁)
        let U₂, d'₂ = if is-exactly(d₂) then extend-record(I \ fields-of(d₂), d₂) else [ ], d₂
        let {fields₁}r₁ = d'₁
        let {fields₂}r₂ = d'₂
        let U₃ = decompose-rows(r₁ ≤ r₂)
        let U₄ = decompose-fields(is-exactly(d₂), fields-of(d₂), U₃(fields₁), U₃(fields₂))
        in U₄ ⊙ U₃ ⊙ U₂ ⊙ U₁

Figure 4.5. Decomposition of Records
decompose-fields : Bool × Label-list × Field-list × Field-list → Substitution

decompose-fields(exactly?, nil, d1, d2) = []
decompose-fields(exactly?, i :: ℓ, d1, d2) =
  let U = decompose-fields(exactly?, ℓ, d1, d2)
  in cases U(d1.i ≤ d2.i) of
    absent ≤ absent or present(v1) ≤ present(v2) ⇒ U
    absent ≤ present(v2) or present(v1) ≤ absent ⇒ failure
    Δ ≤ absent ⇒ [Δ ← absent] o U
    Δ ≤ present(v2) ⇒ [Δ ← present(θ)] o U
    f ≤ Δ2 ⇒ if exactly? then cases f of
    absent ⇒ [Δ2 ← absent] o U
    present(v1) ⇒ [Δ2 ← present(θ)] o U
    Δ1 = U
    else U

Figure 4.5 (Continued)

{A: absent, B: present(v), C: Δ}αexactly ≤ {A: absent, B: present(θ), C: Δ4}βexactly.

We have already seen the result of applying simplify to this constraint.

The following lemma states that the substitution produced by decompose-record-constraint is most general or weakest, i.e., any ground substitution which makes the record constraint d ≤ d' true must factor through the substitution returned by decompose-record-constraint(d ≤ d').

Lemma 4.2 Let $U_e = \text{decompose-record-constraint}(d ≤ d')$, and let $U_0$ be a ground substitution such that $U_0(d ≤ d')$ is true. Then there exists a ground substitution $U'$ such that $U_0 = U' \circ U_e$.

Proof There are two cases:

(i) $d'$ does not have an exactly tag: Then $d$ is extended to have at least those fields that $d'$ has, as is required by the definition of ≤ for both exactly and non-
exactly record types. Function \textit{decompose-fields} then takes over on a record constraint of the form \(\{i : f_i, j : f_j\}r \leq \{i : f_i'\}\beta\), where \(r\) is either \(\alpha\), \(\alpha_{\text{exactly}}\), or \(\text{exactly}\).

We obtain a substitution of the form \(U_e = \Delta_m \rightarrow \text{absent} \mid \Delta_n \rightarrow \text{present}(\theta_n)\), where for all \(m \in M \subseteq I\), \(f_m = \Delta_m\) and \(f'_m = \text{absent}\), and for all \(n \in N \subseteq I\), \(f_n = \Delta_n\) and \(f'_n = \text{present}(v_n)\). By the subtype ordering on type schemes, for each \(n\), \(U_0(\Delta_n) = \text{present}(t_n)\), where \(t_n\) is a ground type. Define \(U'(\theta_n) = t_n\), for all \(n\), and \(U'(\theta) = U_0(\theta)\), otherwise. Now, \(U' \circ U_e(\Delta_m) = \text{absent} = U_0(\Delta_m)\), for all \(m\), \(U' \circ U_e(\Delta_n) = \text{present}(U'(\theta_n)) = \text{present}(t_n) = U_0(\Delta_n)\), for all \(n\), and \(U' \circ U_e(\theta) = U'(\theta) = U_0(\theta)\), otherwise. Thus, \(U_0 = U' \circ U_e\).

(ii) \(d'\) is an exactly record type: Then both \(d\) and \(d'\) are extended to have identical fields as is required by the definition of \(\leq\) (substitution \(U_1\)). Function \textit{decompose-rows} adds an exactly tag to \(d\) if it does not have one (substitution \(U_2\)), since only an exactly record can be a subtype of an exactly record. If only one of \(d\) or \(d'\) has a row variable, it is instantiated to an empty set of fields (substitution \(U_3\)). It is not necessary to maintain the row variable as it could only be extended with "absent" fields. Substitution \(U_0\) must factor through the substitution \(U_0 \circ U_2 \circ U_1\), i.e., there exists a ground substitution \(U'_0\) such that \(U_0 = U'_0 \circ (U_0 \circ U_2 \circ U_1)\).

Function \textit{decompose-fields} takes over on records with the following forms:

\[
\{i : f_i\}_{\text{exactly}} \leq \{i : f'_i\}_{\text{exactly}} \\
\{i : f_i\}_{\alpha_{\text{exactly}}} \leq \{i : f'_i\}_{\beta_{\text{exactly}}}
\]

We obtain a substitution of the form \(U_4 = \Delta_b \rightarrow \text{absent} \mid \Delta_m \rightarrow \text{absent} \mid \Delta_n \rightarrow \text{present}(\theta_n) \mid \Delta_j \rightarrow \text{present}(\theta_j)\), where for all \(b \in B \subseteq I\), \(f_b = \text{absent}\) and \(f'_b = \Delta_b\), for all \(m \in M \subseteq I\), \(f_m = \Delta_m\) and \(f'_m = \text{absent}\), for all \(n \in N \subseteq I\), \(f_n = \text{present}(v_n)\), and \(f'_n = \Delta_n\), and for all \(j \in J \subseteq I\), \(f_j = \Delta_j\) and \(f'_j = \text{present}(v_j)\). By the subtyping
on exactly type record schemes, $U'_o(\Delta_n) = \text{present}(t_n)$ and $U'_o(\Delta_j) = \text{present}(t_j)$, where $t_n, t_j$ are ground types. Define $U'$ as follows: $U'(\theta_n) = t_n$, $U'(\theta_j) = t_j$, and $U'(\theta) = U'_o(\theta)$ otherwise. Now, $U'\circ U_4(\Delta_m) = \text{absent} = U'_o(\Delta_m)$, and $U'\circ U_4(\Delta_n) = \text{present}(U'(\theta_n)) = \text{present}(t_n) = U'_o(\Delta_n)$. Similarly, $U'\circ U_4(\Delta_b) = \text{absent} = U'_o(\Delta_b)$, and $U'\circ U_4(\Delta_j) = \text{present}(U'(\theta_j)) = \text{present}(t_j) = U'_o(\Delta_j)$. Otherwise, $U'\circ U_4(\theta) = U'(\theta) = U'_o(\theta)$. Thus, $U'_o = U'\circ U_4$. $U'$ is the desired substitution, since $U_0 = U'\circ U_4 \circ U_2 \circ U_1 = U'\circ U_o$. □

**Weakness of constraints**

The following lemma states that the substitution $U_o$ produced by $\text{decompose-constraint}(e)$ is *weakest*: any ground substitution $U_0$ such that $U_0(e)$ holds must factor through $U_o$, i.e., $U_0(e)$ is an instance of $U_o(e)$.

**Lemma 4.3** Let $e$ be an element of $C$ that does not have an $M$-form, and let $U_o = \text{decompose-constraint}(e)$. If $U_o \neq \text{failure}$, then for all ground substitutions $U_0$, if $U_0(C)$ is true, then there exists a ground substitution $U'$ such that $U_0 = U' \circ U_o$.

**Proof** Let $U_0$ be a ground substitution such that $U_0(C)$ holds. The proof is by cases on the form of $e$:

(i) $(l : v')\var \leq v$ or $v \leq (l : v')\var$: Function $\text{decompose-constraint}$ unifies the two terms to produce substitution $U_o$. Because the subtyping on $\var$-types requires the terms to be identical, and because unification produces a most general unifier, $U_0$ must factor through substitution $U_o$.

(ii) $v_1 \times v_2 \leq v'_1 \times v'_2$ or $v_1 \rightarrow v_2 \leq v'_1 \rightarrow v'_2$: In both cases, let $U' = U_0$.

(iii) $\theta \leq v_1 \times v_2$ or $\theta \leq v_1 \rightarrow v_2$: In both cases, $U_o = [\theta \mapsto \theta_1 \times \theta_2]$. Since $U_0(e)$ is true, by definition of $\leq$ on $\times$, $U_0$ must instantiate $\theta$ to a product. Thus,
90

\[ U_0(\theta) = t_1 \times t_2, \text{ where } t_1 \text{ and } t_2 \text{ are ground types. Define } U' \text{ as } [\theta_1 \mapsto t_1][\theta_2 \mapsto t_2], \]
and \( U''(\theta') = U_0(\theta') \) for all \( \theta' \) different from \( \theta \). Then \( U''(U_0(\theta)) = U''(\theta_1 \times \theta_2) = t_1 \times t_2 = U_0(\theta) \) and \( U''(U_0(\theta')) = U''(\theta') = U_0(\theta') \), otherwise. Thus, \( U_0 = U' \circ U_2. \)

(iv) \( v_1 \times v_2 \leq \theta \) or \( v_1 \rightarrow v_2 \leq \theta \): Similar to the previous case.

(v) \( s \leq s' \): Follows from Lemma 4.1.

(vi) \( d \leq d' \): Follows from Lemma 4.2.

(vii) \( d \leq \theta \) or \( \theta \leq d \): In both cases, the algorithm instantiates \( \theta \) to \( \{\}\alpha \). By the definition of subtyping, \( U_0 \) must instantiate \( \theta \) to a record type. Since \( \{\}\alpha \) may be unified with any record type, \( U_0 \) must factor through substitution \( [\theta \mapsto \{\}\alpha] \).

The remainder of the proof for this case follows that for \( d \leq d' \). \( \square \)

The following proposition assures us that we can apply algorithm \textit{simplify} to the set of constraints \( U_0(\epsilon) \).

**Proposition 4.2** Let \( U_0 = \text{decompose-constraint}(\epsilon) \), where \( \epsilon \) does not have an \textit{M}-form. If \( U_0 \neq \text{failure} \), then \( U_0(\epsilon) \) has one of the constraint forms that are arguments to function simplify.

The proof of Proposition 4.2 is omitted.

**Partial correctness of decomposition**

We now show the partial correctness of algorithm \textit{decompose}. The following two facts about substitutions and constraints are used in the proofs.

**Fact 4.2** Composition of substitutions is associative.

**Fact 4.3** For all ground substitutions \( U_0 \),

\[ U_0(C_1 \cup C_2) \iff U_0(C_1 \text{ and } C_2) \iff U_0(C_1) \cup U_0(C_2) \iff U_0(C_1) \text{ and } U(C_2) \]
The following two lemmas state that at each iteration $i$, the constraint set $C_i$ is satisfiable if and only if the constraint set $U_i(C_0)$ is satisfiable.

**Lemma 4.4** Assume $\text{decompose}([], C_0)$ terminates successfully after $n$ steps, and that $e_1, \ldots, e_n$ are known. Then, for all $i \in 1..n$, for all ground substitutions $U$, $U(C_i)$ implies $U(U_i C_0)$.

**Proof**

(Basis) Trivial, since $U_0 = []$.

(Induction) Assume $U(C_i)$ holds. $U(C_i) = U(C_{e_i} \cup U_{e_i}(C_{i-1} - \{e_i\}))$, by definition of $C_i$. Since $U \circ U_{e_i}$ is ground, by Facts 4.2 and 4.3, this equals

\[(*) \quad U(C_{e_i}) \text{ and } U \circ U_{e_i}(C_{i-1} - \{e_i\})\]

By definition of $\text{decompose}$, $C_{e_i} = \text{simplify}(U_{e_i} e_i)$. Since $U$ is ground, by Proposition 4.1, $U(\text{simplify}(U_{e_i} e_i))$ implies $U(U_{e_i} e_i)$. By Fact 4.2, this equals $U \circ U_{e_i}(e_i)$.

By replacing the left-hand term of $(*)$, we obtain $U \circ U_{e_i}(e_i)$ and $U \circ U_{e_i}(C_{i-1} - \{e_i\})$.

By Fact 4.3, this equals $U \circ U_{e_i}(e_i \cup (C_{i-1} - \{e_i\}))$, which equals $U \circ U_{e_i}(C_{i-1})$. Since $U \circ U_{e_i}$ is ground, by the induction hypothesis this implies $U \circ U_{e_i}(U_{i-1} C_0) = true$.

By Fact 4.2, $U(U_{e_i} \circ U_{i-1}(C_0)) = true$, and since $U_{e_i} \circ U_{i-1} = U_i$, we obtain $U(U_i C_0) = true$. □

**Lemma 4.5** Assume $\text{decompose}([], C_0)$ terminates successfully after $n$ steps, and that $e_1, \ldots, e_n$ are known. Then, for all $i \in 1..n$, for all ground substitutions $U$, $U(U_i C_0)$ implies $U(C_i)$.

**Proof**

(Basis) Trivial, since $U_0 = []$. 
(Induction) Assume \( U(U_{i}, C_{0}) \) holds. By definition, \( U_{i} = U_{e_{i}} \circ U_{i-1} \). Since composition of substitutions is associative (Fact 4.2), \( U \circ U_{e_{i}}(U_{i-1}C_{0}) = true \). By the inductive hypothesis, since \( U \circ U_{e_{i}} \) is ground, this implies \( U \circ U_{e_{i}}(C_{i-1}) = true \). Now rewrite \( C_{i-1} \) to obtain \( U \circ U_{e_{i}}(({C_{i-1} - \{e_{i}\}}) \cup e_{i}) \). By Fact 4.3, this equals

\[
(*) \quad U \circ U_{e_{i}}(C_{i-1} - \{e_{i}\}) \quad \text{and} \quad U \circ U_{e_{i}}(e_{i})
\]

By Fact 4.2 and Proposition 4.1, \( U \circ U_{e_{i}}(e_{i}) = U(U_{e_{i}}(e_{i})) \) implies \( U(\text{simplify}(U_{e_{i}}(e_{i}))) \).

By the definition of \( \text{decompose} \), this is equal to \( U(C_{e_{i}}) \).

Now replace the right-hand term of \( (*) \) to obtain \( U \circ U_{e_{i}}(C_{i-1} - \{e_{i}\}) \) and \( U(C_{e_{i}}) \).

By Fact 4.2, this equals \( U(U_{e_{i}}((C_{i-1} - \{e_{i}\})) \cup C_{e_{i}}) \). By Fact 4.3, this equals \( U(U_{e_{i}}((C_{i-1} - \{e_{i}\})) \cup C_{e_{i}}) \). And by definition of \( C_{i} \), we obtain \( U(C_{i}) \).

The following lemma states that at each iteration step \( i \), if the constraint set \( C_{i-1} \) is satisfiable, then the constraint set \( C_{i} \) is satisfiable.

**Lemma 4.6** Let \( C_{0} \) be the initial set of constraints. Assume \( \text{decompose}([], C_{0}) \) terminates successfully after \( n \) steps, and that \( e_{1}, \ldots, e_{n} \) are known. Then for all \( i \in 1..n \), for all ground substitutions \( U \), \( U(C_{i-1}) \) implies there exists a ground substitution \( U' \) such that

\[
(a) \quad U = U' \circ U_{e_{i}}
\]

\[
(b) \quad U'(C_{i})
\]

**Proof**

(a) Follows from Lemma 4.3.

(b) Let \( U \) be a ground substitution such that \( U(C_{i-1}) \) holds. From (a), let \( U = U' \circ U_{e_{i}} \), thus \( U' \circ U_{e_{i}}(C_{i-1}) \) holds. By Lemma 4.4, this implies \( U' \circ U_{e_{i}}(U_{i-1}C_{0}) \). By
Fact 4.2, this equals $U'(U_e \circ U_{i-1}(C_0))$. By definition of $U_i$, this equals $U'(U_i(C_0))$.

By Lemma 4.5, this implies $U'(C_1)$. □

The following theorem states the partial correctness properties for algorithm `decompose`. When `decompose([], C_0)` terminates successfully with result constraint set $C_n$ and substitution $U_n$, the constraint sets $C_n$, $U_n(C_0)$, and $C_0$ are “equivalent” in the sense that if one of the sets is satisfiable (not satisfiable), then so are the others. Thus, we may use the constraint sets $C_n$, $U_n(C_0)$, and $C_0$ interchangeably.

**Theorem 4.1** Let $C_0$ be the initial set of constraints. Assume `decompose([], C_0)` terminates successfully after $n$ steps. Let $U_n, C_n = decompose([], C_0)$.

(a) For all ground substitutions $U$, $U(C_0)$ implies there exists a ground substitution $U'$ such that $U = U' \circ U_n$

(b) is-satisfiable($C_n$) $\iff$ is-satisfiable($U_n C_0$)

(c) is-satisfiable($U_n C_0$) $\iff$ is-satisfiable($C_0$)

**Proof**

(a) Let $S_0$ be a ground substitution such that $S_0(C_0)$. By repeated applications of Lemma 4.6, there exist ground substitutions $S_1, S_2, \ldots, S_n$ such that

\[
S_0 = S_1 \circ U_{e_1} \quad \text{and} \quad S_1(C_1)
\]

\[
S_1 = S_2 \circ U_{e_2} \quad \text{and} \quad S_2(C_2)
\]

\[\vdots\]

\[
S_{n-1} = S_n \circ U_{e_n} \quad \text{and} \quad S_n(C_n)
\]

Replacing equals by equals, using Fact 4.2, $S_0 = S_1 \circ U_{e_1} = S_2 \circ U_{e_2} \circ U_{e_1} = \cdots = S_n \circ U_{e_n} \circ U_{e_{n-1}} \circ \cdots \circ U_{e_2} \circ U_{e_1}$ By the definition of $U_n$, this equals $S_n \circ U_n$. Thus, $S_n$ is the desired substitution.
(b) Follows from Lemmas 4.4 and 4.5.

(c) Let \( U \) be a ground substitution such that \( U(U_nC_0) \) holds. By Fact 4.2, 
\( U \circ U_n(C_0) \) holds. Thus, \( \text{is-satisfiable}(U_nC_0) \) implies \( \text{is-satisfiable}(C_0) \).

Let \( U \) be a ground substitution such that \( U(C_0) \) holds. From part (a), \( U(C_0) \) implies there exists a ground substitution \( U' \) such that \( U = U' \circ U_n \). By Fact 4.2, 
\( U'(U_nC_0) \) holds. Thus, \( \text{is-satisfiable}(C_0) \) implies \( \text{is-satisfiable}(U_nC_0) \). \( \square \)

**Satisfaction of distinguished constraints**

Once *decompose* has been applied to an initial set of constraints \( C_0 \) to obtain the set of decomposed constraints \( C_M \), the remaining constraints all have M-forms. We now remove the record and storage constraints from \( C_M \) to obtain a set of atomic constraints. Fuh and Mishra's algorithm CONSISTENT is then applied to the set of atomic constraints to (finally!) test the satisfiability of \( C_0 \). Before showing how to remove the record and storage constraints from \( C_M \), we introduce some additional rewrite rules for grounding type schemes.

**Disappearing deltas** In addition to "absent" and "present" fields, we have need for a third form of instantiation into field variables. Consider the following constraints:

\[
\{B : \text{absent}\}_{\text{exactly}} \leq \{B : \Delta_B\} \beta \text{ and } \{B : \text{present}(v_B)\} \alpha \leq \{B : \Delta_B\} \beta
\]

In order to make this constraint set hold true, we must instantiate \( \beta \) to the empty record. If an exactly tag were added, this would force \( \alpha \) to be extended with an exactly tag as well. This in turn would force \( \Delta_B \) to be instantiated to a present field
to satisfy the second constraint. However, this would cause the first constraint to become false.

We would like some way to make the $B$ field “disappear” from the record $\{B : \Delta_B\} \beta$. We introduce a new field called don’t-care for this purpose. A field variable may be instantiated with don’t-care to obtain a ground type scheme, and then eliminated with the following rules to obtain a type:

\[
\begin{align*}
(E-v) & \quad \{\ldots I: \text{don't-care}\ldots\} \text{exactly} \Rightarrow \{\ldots\} \text{exactly} \\
(E-vi) & \quad \{\ldots I: \text{don't-care}\ldots\} = \{\ldots\}
\end{align*}
\]

Elimination of record and storage constraints Let $U$ be a substitution that instantiates all storage type variables to $\{\}$, all field variables to don’t-care, and all row variables to $\{\}$. We eliminate the record and storage constraints from the set of M-form constraints by applying substitution $U$ to them. $U$ causes the constraints to become trivially satisfied and allows us to eliminate them from the set of M-forms, leaving a set of atomic constraints. The next proposition proves this property of substitution $U$.

**Proposition 4.3** Let $C$ be a set of constraints with M-forms. Let $U$ be a substitution which instantiates all storage type variables to $\{\}$, all row variables to $\{\}$, and all field variables to don’t-care. Application of $U$ to $C$ makes all constraint forms in $U(C)$ trivially satisfied.

**Proof** The proof is by cases on the form of $\epsilon$:

(i) $s \leq \langle \rangle \sigma$: Then $\sigma$ disappears and $U(s) \leq \langle \rangle$ holds for any instantiation of the type variables in $U(s)$. 
(ii) \( \{i : f_i, j : f_j\}^r \leq \{i : \Delta_i\}^\beta : \) Then \( \beta \) disappears and each \( \Delta_i \) is instantiated to don't-care. The right-hand side is now equal to \( \{i : \text{don't-care}\} \). By the equivalence rewrite rules, this is equal to the empty record. Thus, the constraint trivially holds true for any instance of the left-hand side.

(iii) \( \{i : \Delta_i\}^{\alpha_{\text{exactly}}} \leq \{i : \Delta_i\}^{\beta_{\text{exactly}}} : \) Application of the substitution to the constraint gives the scheme \( \{i : \text{don't-care}\}^{\text{exactly}} \leq \{i : \text{don't-care}\}^{\text{exactly}} \). By the equivalence rewrite rules, the constraint is equal to \( \{\}^{\text{exactly}} \leq \{\}^{\text{exactly}} \), which is trivially true.

(iv) \( \{i : \Delta_i\}^{\text{exactly}} \leq \{i : \Delta_i\}^{\text{exactly}} : \) Same as the previous case. □

Rowlessness and invisibility

In the next chapter, we introduce constraints on record and storage type schemes whose right-hand term does not contain a row or storage variable. The constraints have forms such as:

\[
\{B: \text{present}(\text{int}), C: \Delta\}^{\alpha_3} \leq \{D: \text{present}(\theta)\}
\]

\[
\langle \ell_2: \text{int}, \ell_6: \text{int}\rangle^{\sigma_4} \leq \langle \ell_3: \theta_2\rangle^{\sigma_35}
\]

which are not generated by the syntax of type schemes in Figure 3.1.

We can use algorithm decompose without modification for these constraint forms by adding a fresh row or storage variable to the right-hand scheme. For example, the fresh variables \( \alpha_27 \) and \( \sigma_{35} \) can be added to the above type schemes to obtain the constraints: \( \{B: \text{present}(\text{int}), C: \Delta\}^{\alpha_3} \leq \{D: \text{present}(\theta)\}^{\alpha_27} \) and \( \langle \ell_2: \text{int}, \ell_6: \text{int}\rangle^{\sigma_4} \leq \langle \ell_3: \theta_2\rangle^{\sigma_{35}} \). Because the variables do not appear in any other constraint, and because the row variable does not have an “exactly” tag, they will not be extended with other fields during decomposition. Thus the new variables are in a sense “invisible.”
CHAPTER 5. APPLICATION AND IMPLEMENTATION

In this chapter we define the action semantics for a small imperative language with constant, macro, and procedure declarations, and stack-like variable allocation and deallocation. We simplify the language of types presented in Chapter 2 by removing subtyping constraints on the functional facet. We define actions for making dynamically scoped expression abstracts (macros) and statically scoped command abstracts (procedures). These actions make use of the decomposition algorithms presented in the previous chapter. We show the details of the type inference algorithms on a small program—a tedious task by hand! Finally, we discuss the ML implementation of the type inference algorithms.

Specialized Action Semantics

Facets

The types we will be using are given in Figure 5.1. The primitive types are now restricted to the single type \texttt{int}. \texttt{cmd} is the type of a command abstract, parameterized on the shape of storage required for its execution. \texttt{exp} is the type of an expression abstract, parameterized on the set of bindings and the shape of storage required for its evaluation.

The declarative and imperative facets are defined as before, but the types in the
imperative facet are made simpler now as the only values to be stored in a cell will be of type `int`.

The subtype ordering on the types in the functional facet is the reflexive ordering, plus $t \leq ns$, for all $t$. The subtype orderings on the types in the declarative and imperative facets are the same as those given in Chapter 2. The joins listed in the figure are a consequence of that ordering. We state the joins due to their importance to the actions in Figure 5.4.

The interpretation of the types in the functional facet is given in Figure 5.2. A `cmds` abstract is a function that acts upon an imperative facet argument of type $s$ without changing its shape, i.e., it does not allocate or deallocate cells. An `exp(d, s)` abstract is an integer-valued expression, interpreted as a function from a set of bindings of type $d$ and a set of memory cells of type $s$ to a value of type `int`.

The interpretations of the declarative and imperative facets are the same as those given in Chapter 2.

**Primitive actions**

For the type schemes for the actions, we introduce field, row, location, and storage variables as before, but we will not have type variables that range over the functional facet. Because of this, we index the primitive actions, where necessary, with the form of argument or result required from the functional facet. The schemes for these specialized primitive actions are given in Figure 5.3. The only "forget" actions that involve the functional facet which are needed for our example language are the three listed. The forget actions that only involve the declarative and imperative facets do not require modification of their typing schemes.
SYNTAX

\[ t \in \text{Proper-functional-type} \]
\[ d \in \text{Proper-declarative-type} \]
\[ s \in \text{Proper-imperative-type} \]
\[ t ::= \text{int} \mid t_1 \times t_2 \mid \text{cmds} \mid \ell \text{var} \mid \text{exp}(d, s) \]
\[ d ::= \{ i : t_i \}_{i \in I} \mid \{ i : t_i \}_{i \in I} \text{exactly} \]
\[ s ::= (\ell : \text{int})_{\ell \in L} \]

SUBTYPING

\[ t \leq ns, \text{ for all } t \]
\[ d \leq ns, \text{ for all } d \]
\[ \{ i : t_i \}_{i \in I} \text{exactly} \leq \{ i : t_i \}_{i \in I} \]
\[ \{ i : t_i \}_{i \in I} \leq \{ j : t_j \}_{j \in J}, \text{ if and only if } J \subseteq I \]
\[ s \leq ns, \text{ for all } s \]
\[ (k : \text{int})_{k \in K} \leq (\ell : \text{int})_{\ell \in L}, \text{ if and only if } L \subseteq K \]

JOINS

\[ \{ i : t_i \}_{i \in I} \cup \{ j : t_j \}_{j \in J} = \{ k : t_k \}_{k \in K \cap J} \]
\[ \{ i : t_i \}_{i \in I} \text{exactly} \cup \{ j : t_j \}_{j \in J} = \{ k : t_k \}_{k \in K \cap J} \]
\[ \{ i : t_i \}_{i \in I} \text{exactly} \cup \{ j : t_j \}_{j \in J} \text{exactly} = \{ k : t_k \}_{k \in K \cap J} \]
\[ (k : \text{int})_{k \in K} \cup (\ell : \text{int})_{\ell \in L} = (m : \text{int})_{m \in K \cap L} \]

Figure 5.1. Specialized Facets
\( A_F[\text{int}] = \mathbb{Z} \)
\( A_F[t_1 \times t_2] = A_F[t_1] \times A_F[t_2] \)
\( A_F[\text{var}] = (A_I[\ell] \to A_F[\text{int}]) \times (A_F[\text{int}] \to A_I[\ell]) \)
\( A_F[\text{cmds}] = A_I[s] \to A_I[s] \)
\( A_F[\text{exp}(d, s)] = A_D[d] \times A_I[s] \to A_F[\text{int}] \)

Figure 5.2. Interpretation of the Functional Facet

ACTION SCHEME

<table>
<thead>
<tr>
<th>ACTION</th>
<th>SCHEME</th>
</tr>
</thead>
<tbody>
<tr>
<td>put\text{int} n</td>
<td>1 \rightarrow \text{int}</td>
</tr>
<tr>
<td>add\text{int}</td>
<td>\text{int} \times \text{int} \rightarrow \text{int}</td>
</tr>
<tr>
<td>bind\text{int} i</td>
<td>\text{int} \rightarrow {i : \text{present}(\text{int})} \text{exactly}</td>
</tr>
<tr>
<td>bind\text{cmd} i</td>
<td>\text{cmd}(\ell) \sigma \rightarrow {i : \text{present}(\text{cmd}(\ell))} \text{exactly}</td>
</tr>
<tr>
<td>bind\text{var} i</td>
<td>i_{\text{var}} \rightarrow {i : \text{present}(i_{\text{var}})} \text{exactly}</td>
</tr>
<tr>
<td>bind\text{exp} i</td>
<td>\text{exp}({\alpha, (\sigma)}) \rightarrow {i : \text{present}(\text{exp}({\alpha, (\sigma)}))} \text{exactly}</td>
</tr>
<tr>
<td>find\text{int} i</td>
<td>{i : \text{present}(\text{int})} \alpha \rightarrow \text{int}</td>
</tr>
<tr>
<td>find\text{cmd} i</td>
<td>{i : \text{present}(\text{cmd}(\ell))} \alpha \rightarrow \text{cmd}(\ell)</td>
</tr>
<tr>
<td>find\text{var} i</td>
<td>{i : \text{present}(i_{\text{var}})} \alpha \rightarrow i_{\text{var}}</td>
</tr>
<tr>
<td>find\text{exp} i</td>
<td>{i : \text{present}(\text{exp}({\beta, (\sigma)}))} \alpha \rightarrow \text{exp}({\beta, (\sigma)})</td>
</tr>
<tr>
<td>alloc\text{int}</td>
<td>(\ell : \text{undef}) \sigma \rightarrow \ell_{\text{var}} \times (\ell : \text{int}) \sigma \quad \text{where } \ell = \text{newlocn}()</td>
</tr>
<tr>
<td>contents</td>
<td>i_{\text{var}} \times (i : \text{int}) \sigma \rightarrow \text{int}</td>
</tr>
<tr>
<td>update</td>
<td>(i_{\text{var}} \times (i : \text{int})) \times (i : \text{int}) \sigma \rightarrow (i : \text{int}) \sigma</td>
</tr>
<tr>
<td>forget_{FD,F}</td>
<td>i_{\text{var}} \times {} \alpha \rightarrow i_{\text{var}}</td>
</tr>
<tr>
<td>forget_{FD,I}</td>
<td>i_{\text{var}} \times {} \alpha \times (\sigma) \rightarrow (\sigma)</td>
</tr>
<tr>
<td>forget_{FD,FI}</td>
<td>i_{\text{var}} \times {} \alpha \times (\sigma) \rightarrow i_{\text{var}} \times {} \alpha</td>
</tr>
</tbody>
</table>

Figure 5.3. Specialized Schemes for Primitives
Command and expression abstracts

In order to define the semantics for language constructs such as procedures, we need to define actions for placing a command denotation on the functional facet and enacting it later. The actions for doing this are specified in Figure 5.4. *declare* makes a statically-scoped command abstract, or procedure, and *call* enacts the command abstract later, provided that the storage cells it requires are available in the context of the call. *define* makes a dynamically-scoped expression abstract, or macro definition, and *eval* enacts the expression abstract later, provided that the bindings and storage cells it requires are available in the context where it is evaluated.

Because an action is a natural transformation, which is a family of functions, we cannot put an entire action onto the functional facet. Instead, we must select a single morphism from the action to output. To do this, we choose a "best" type
MEANING FUNCTION

\[ A_{\text{declare}}(d) = \lambda r : d . \lambda p : s_0 . a(d, s_0)(r, p) \]

where \( d_0 \times s_0 = \sqcup \text{domain}(T_a) \)

\[ A_{\text{call}}(\text{cmd} s_0, s) = \lambda (c : \text{cmd} s_0, p : s) . (A_I[s \leq s_0] \downarrow 2) c p \]

\[ A_{\text{define}}(d) = \lambda () . a(d_0, s_0) \]

where \( d_0 \times s_0 = \sqcup \text{domain}(T_a) \)

\[ A_{\text{eval}}(\text{exp}(d_0, s_0), d, s) = \]

\[ \lambda (f : \text{exp}(d_0, s_0), r : d, p : s) . f(A_D[d \leq d_0] r, (A_I[s \leq s_0] \downarrow 1) p) \]

Figure 5.4 (Continued)

from the domain of the action's typing function. For an action \( a : K_1 \rightarrow K_2 \), say that \( \text{domain}(T_a) = \{ t \in K_1 \mid T_a(t) \neq \text{ns} \} \). We compute the best type for an action \( a \) by taking the least upper bound of \( \text{domain}(T_a) \). In order for the least upper bound to be worthwhile, we require that for all actions \( a \), \( T_a(\sqcup \text{domain}(T_a)) \neq \text{ns} \).

Now our reason for restricting the functional facet to the single primitive type \text{int} and introducing specialized versions of the primitive actions becomes clear. The primitive actions are now essentially monomorphic on the functional facet. However, the actions still exhibit forms of polymorphism on both the declarative and imperative facets. The "forget" actions exhibit parametric polymorphism over an entire facet, but both facets have a best type: the empty record type \{ \} in the declarative facet, and the empty storage type \{ \} in the imperative facet. All of the other primitive actions exhibit inclusion polymorphism on these facets and by definition have a best type.

Although the abstraction actions \text{declare} and \text{define} cut down an action \( a \) into a single morphism, the actions \text{call} and \text{eval} recover the polymorphism in \( a \) when
they enact it later. *call* applies a command abstract \( c \) with type \( \text{cmd}s_0 \) to a set of storage cells \( p \) of type \( s \leq s_0 \) by first lifting the command abstract from a function on storage of shape \( s_0 \) to a function on storage of shape \( s \). The abstracted action can thus be applied to any shape storage which it could have acted upon before it was abstracted.

*eval* applies an integer-typed expression abstract \( f \) with type \( \text{exp}(d_0, s_0) \) to a set of bindings \( r \) of type \( d \leq d_0 \) and a set of storage cells \( p \) of type \( s \leq s_0 \) by first applying a coercion to each argument. Here we see a crucial use of the inheritance typing. If \( d_0 \) is an "exactly" type, the abstract may only be enacted in contexts that have exactly the bindings in \( d_0 \). However, because of the non-exact record types, if \( d_0 \) does not have an exactly type, the abstract can be enacted in any context that has at least the bindings in \( d_0 \).

Note that for both forms of abstracts we have restricted the target of the action \( a \) so that we can know exactly what form the target of the abstracted version of \( a \) should have when it is enacted on its arguments: command abstracts require that \( a \) have the same shape storage in its source and target; expression abstracts require that \( a \) produce a value of type \text{int} as its result.

In [9], we defined statically-scoped functions on a single argument using the action \( \text{freeze}_{t_0}a \), where \( a : F D \rightarrow F \). Suppose action \( a \) can be applied to arguments of type \( t \leq \text{real} \) and produces an answer of type \( t \). If \( t_0 = \text{real} \), action \( a \) is converted into a function on values of type \text{real}. When the abstracted action is later enacted, *eval* recovers only some of the polymorphism in \( a \): it may be applied to arguments of type \( t' \leq \text{real} \), but when \( t' \) is \text{int}, the result will have type \text{real} instead of type \text{int} as it would have had for the original action \( a \). This is a limitation of our approach.
The schemes for the actions in Figure 5.4 are given in Figure 5.5. It is easy for the type inference on \textit{declare a} to check the equality of $T\alpha(d,s)$ and $s$ using the typing scheme inferred for $a$. Function \textit{analyze} is the type inference algorithm itself. The inference makes use of the function \textit{fill-in-storage} which, when applied to a storage type scheme, returns a substitution that instantiates its storage variable to $\{\}$, together with the resulting scheme, and the function \textit{fill-in-scheme} which is defined as:

\[
\text{fill-in-scheme}(S_1 \rightarrow S_2 \text{ if } C) = \\
\begin{cases} 
\text{if } U_M C_M = \text{decompose}([], C) \neq \text{failure} \text{ and } U_G \circ U_M(C) = \text{true} \\
\text{then } U_G \circ U_M(S_1) \rightarrow U_G \circ U_M(S_2) \text{ else ns } \rightarrow \text{ns if false}
\end{cases}
\]

where $U_G$ is the substitution that instantiates all row variables to $\{\}$, all storage variables to $\{\}$, and all field variables to \textit{don't-care}. By Proposition 4.3, the constraints in $U_G \circ U_M(C)$ are trivially satisfiable.

Because constraints on record type schemes and storage type schemes appear in the schemes for these actions, we will see the application of the algorithms presented
I ∈ Identifier
E ∈ Expression
M ∈ Allocation
D ∈ Declaration
C ∈ Command

E ::= N | E₁ + E₂ | I | use I | expand I
M ::= var I | M₁ ; M₂
D ::= const I = E | proc I = C | macro I = E | D₁ and D₂
C ::= I := E | begin D in C end | alloc M in C free | C₁ ; C₂ | call I

Figure 5.6. An Example Language Definition: Syntax

in the last chapter in the example of type inference in the next section.

We may define other forms of abstracts such as modules (declaration abstracts) in a similar manner, by defining actions for abstracting and enacting an action that produces results in the declarative facet.

An example language definition

Figure 5.6 specifies the abstract syntax for a small imperative language. The language has constant, procedure, and macro declarations, declaration blocks, and stack-like variable allocation and deallocation. Variable allocations are made a separate syntactic construct from the other declarations, and their use is limited in scope to the command C in the phrase alloc M in C free.

The action semantics for the language is given in Figure 5.7. The definitions look slightly more complex than necessary. Rather than use the derived combinator, we have written in all of the necessary forget actions to make the example of type inference we present later easier to follow.
\( \mathcal{E} : \text{Expression} \rightarrow \text{Action}_{P}^{D} \)

\[ \mathcal{E}[N] = \text{forget}_{(D,I,D)} ; \text{put} \_N \]  

\[ \mathcal{E}[I] = \text{forget}_{(D,I,D)} \; \text{find} \_\text{int} \]  

\[ \mathcal{E}[\text{use } I] = ((\text{forget}_{(D,I,D)} \; \text{find} \_\text{var} I) * \text{forget}_{(D,I,D)}) ; \text{contents} \]  

\[ \mathcal{E}[\text{expand } I] = ((\text{forget}_{(D,I,D)} \; \text{find} \_\text{exp} I) * \text{forget}_{(D,I,D)}) ; \text{eval} \]  

\[ \mathcal{E}[E_1 + E_2] = (\mathcal{E}[E_1] * \mathcal{E}[E_2]) ; \text{add}_{\text{int}} \]  

\[ \mathcal{M} : \text{Allocation} \rightarrow \text{Action}_{P}^{D} \]  

\[ \mathcal{M}[M_1 ; M_2] = \mathcal{M}[M_1] ; \mathcal{M}[M_2] \]  

\[ \mathcal{M}[\text{var } I] = ((\text{forget}_{(D,I,D)} \; \text{alloc}_{\text{int}}) * \text{forget}_{(D,I,D)}) ; \]  

\[ ((\text{forget}_{(F, D, I, F)} ; \triangleright (\text{forget}_{(F, D, F)} ; \text{bind} \_\text{var} I)) * \text{forget}_{(F, D, I, F)}) \]  

\[ \mathcal{D} : \text{Declaration} \rightarrow \text{Action}_{P}^{D} \]  

\[ \mathcal{D}[\text{const } I = E] = \mathcal{E}[E] ; \text{bind}_{\text{int}} \]  

\[ \mathcal{D}[\text{proc } I = C] = (\text{forget}_{(D,I,D)} ; \text{declare } C[C]) ; \text{bind}_{\text{cmd}} \]  

\[ \mathcal{D}[\text{macro } I = E] = (\text{forget}_{(D,I,D)} ; \text{define } E[E]) ; \text{bind}_{\text{exp}} \]  

\[ \mathcal{D}[D_1 \text{ and } D_2] = D[D_1] * D[D_2] \]  

\[ \mathcal{C} : \text{Command} \rightarrow \text{Action}_{P}^{D} \]  

\[ \mathcal{C}[C_1 ; C_2] = (\mathcal{C}[C_1] * \text{forget}_{(D,I,D)}) ; \mathcal{C}[C_2] \]  

\[ \mathcal{C}[I := E] = ((\text{forget}_{(D,I,D)} ; \text{find} \_\text{var} I) * \text{forget}_{(D,I,D)} * E[E]) ; \text{update} \]  

\[ \mathcal{C}[\text{begin } D \text{ in } C \text{ end}] = (\triangleright (D[D]) * \text{forget}_{(D,I,D)}) ; \mathcal{C}[C] \]  

\[ \mathcal{C}[\text{alloc } M \text{ in } C \text{ free}] = \mathcal{M}[M] \text{ in } \mathcal{C}[C] \]  

\[ \mathcal{C}[\text{call } I] = ((\text{forget}_{(D,I,D)} ; \text{find} \_\text{cmd} I) * \text{forget}_{(D,I,D)}) ; \text{call} \]  

Figure 5.7. An Example Language Definition: Action Semantics
Notice that identifiers used as expressions have typing annotations as part of their syntax. Alternatively, we could have combined the denotations for the three forms of evaluation into a single equation for $E[I]$ by using the conditional combinator \("/\) in its definition.

Declarations are accumulated using parallel composition. This has the effect that duplicate declarations in the same block cause a typing error. However, when a declaration block $D$ occurs in the context \textbf{begin} $D$ \textbf{in} $C$ \textbf{end}, its bindings override bindings occurring in an enclosing scope. In contrast, because variable allocations are done sequentially, we have chosen to accumulate variable declarations using sequential composition, with a new declaration overriding the existing bindings as it is processed. Thus, duplicate variable declarations do not induce a typing error. Variable deallocation is handled by a new combinator called \textit{in} that we describe in the next section.

\textbf{The IN combinator}

The combinator \textit{in} is used to limit the scope of variable allocations. The action $a_1$ in $a_2$ deallocates those locations allocated in action $a_1$ upon completion of action $a_2$. This combinator is not as general as the other combinators as we are defining it for a specific application.

For actions $a_1 : DI \rightarrow DI$ and $a_2 : DI \rightarrow I$, we define the typing function for $a_1$ in $a_2$ as:

$$T_{a_1 \text{ in } a_2}(d, s) = s$$

if $T_{a_2} \circ T_{a_1}(d, s) \neq \text{ns}$

and $\forall d', \forall s'$, if $T_{a_2}(d', s') \neq \text{ns}$, then $T_{a_2}(d', s') = s'$
Action $a_2$ is restricted to produce results only in the imperative facet so that upon its completion we can truncate the storage to the shape it had when given to $a_1$. We want to be sure that any variables that are being deallocated are not being passed along in another facet where they could still be used. We have also restricted $a_2$ to have the same shape storage in its source and target. The creation of command abstracts is made easier when commands do not alter the shape of the store. In our use of the combinator $in$ in the imperative language we define, its second argument is always a command denotation.

We define the meaning function for $a_1 in a_2$ as:

$$A_{a_1 in a_2} (d, s) = \lambda (r: d, p: s). \\
(A_f [T_{a_2} \circ T_{a_1} (d, s) \leq s] \downarrow 1) (a_2 (T_{a_1} (d, s)) (a_1 (d, s) (r, p)))$$

The coercion function is applied to the result storage to remove the cells allocated by $a_1$. When $T_{a_2} \circ T_{a_1} (d, s) \neq ns$, we know that the subtyping relation $T_{a_2} \circ T_{a_1} (d, s) \leq s$ holds because none of our actions remove cells from their imperative facet arguments.

For actions $a_1: DI \leadsto DI$ with scheme $\$_1$, and $a_2: DI \leadsto I$ with scheme $\$_2$, we define the type inference for $a_1 in a_2^{in(\$_1, \$_2)}$ as:

$$in(\$_1 \leadsto \$_2 \mbox{ if } C, \$'_1 \leadsto \$'_2 \mbox{ if } C') = \\
U(\$_1) \leadsto U(\mbox{imperative-facet-in}(\$_1)) \mbox{ if } U(C \mbox{ and } C')$$

where $U = \mbox{unify}(\$_2, \$'_1) \neq \mbox{failure}$

and $U(\mbox{imperative-facet-in}(\$'_1)) = U(\$'_2)$

$$\mbox{ns} \leadsto \mbox{ns} \mbox{ if false, otherwise}$$

As for the action $\mbox{declare}$, the type inference algorithm can easily test the equality of $T_{a_2} (d, s)$ and $s$ for all non-$\mbox{ns}$ results by using the typing schemes.
begin
  macro A = use C
  and
  proc P = X := use X + use C

begin
  call P;
  alloc
  var C
  in
    C := expand A;
    call P
  free
end

Figure 5.8. The Example Program

An example of type inference

We will now show the type inference for the program in Figure 5.8. First, we
consider the outer declaration block. The type inference for

\[ a_1 = \mathcal{E}[\text{use } C] = (((\text{forget}_{D\cdot D}) \circ \text{find}_{\text{var } C}) \circ \text{forget}_{D\cdot I},) \circ \text{contents} \]

infers the following schemes for the primitives actions:

\[
\begin{align*}
\text{forget}_{D\cdot D} &\in \{ } & & \alpha_1 \times \langle \rangle \sigma_1 &\mapsto \{ } & & \alpha_1 \\
\text{find}_{\text{var } C} &\in \{ C : \iota_1 \text{var} \} & & \alpha_2 &\mapsto \iota_1 \text{var} \\
\text{forget}_{D\cdot I} &\in \{ } & & \alpha_3 \times \langle \rangle \sigma_2 &\mapsto \langle \rangle \sigma_2 \\
\text{contents} &\in \iota_2 \text{var} \times \langle \iota_2 : \text{int} \rangle \sigma_3 &\mapsto \text{int}
\end{align*}
\]

Unification of the target scheme of \text{forget}_{D\cdot D} with the source scheme of \text{find}_{\text{var } C}
produces the substitution \([\alpha_1 \mapsto \{ C : \Delta_1 \} \alpha_4][\alpha_2 \mapsto \{ } \alpha_4][\Delta_1 \mapsto \iota_1 \text{var}], which is
applied to the source and target schemes to obtain:
\((\text{forget}_{\text{DI,1}} ; \text{find}_\text{var} C) \in \{ C : \iota_1 \text{var}\} \alpha_4 \times \langle \rangle \sigma_1 \rightharpoondown \iota_1 \text{var} \)

This action is then composed in parallel with \(\text{forget}_{\text{DI,1}}\). The type inference unifies the two source schemes to produce the substitution \([\alpha_3 \mapsto \{ C : \Delta_1\}\alpha_5][\alpha_4 \mapsto \{ \} \alpha_5][\Delta_2 \mapsto \iota_1 \text{var}][\sigma_1 \mapsto \langle \rangle \sigma_4][\sigma_2 \mapsto \langle \rangle \sigma_4]\), which is applied to the source and target schemes. The targets are then merged to obtain:

\(((\text{forget}_{\text{DI,1}} ; \text{find}_\text{var} C) * \text{forget}_{\text{DI,1}}) \in \{ C : \iota_1 \text{var}\} \alpha_5 \times \langle \rangle \sigma_4 \rightharpoondown \iota_1 \text{var} \times \langle \rangle \sigma_4\)

Finally, this action is composed sequentally with \(\text{contents}\). Unification produces the substitution \([\iota_1 \mapsto \iota_3][\iota_2 \mapsto \iota_3][\sigma_3 \mapsto \langle \iota_3 : \text{int}\rangle \sigma_5][\sigma_4 \mapsto \langle \rangle \sigma_5]\), which is applied to the source and target schemes to obtain:

\(a_1 \in \{ C : \iota_3 \text{var}\} \alpha_5 \times \langle \iota_3 : \text{int}\rangle \sigma_5 \rightharpoondown \text{int}\)

Because the "forget" actions are used simply to pass along facets, we will omit the details of the type inference for these actions from now on unless it is needed for clarity. The type inference for the declaration

\(a_2 = D[\text{macro } A = \text{use } C] = (\text{forget}_{\text{DI,1}} ; \text{define } a_1) ; \text{bind}_\text{exp}A\)

infers the following schemes:

\((\text{forget}_{\text{DI,1}} ; \text{define } a_1) \in \{ \} \alpha_6 \times \langle \rangle \sigma_6 \rightharpoondown \exp(\{ C : \iota_3 \text{var}\}, \langle \iota_3 : \text{int}\})\)

\(\text{bind}_\text{exp}A \in \exp(\{ \} \alpha_7, \langle \rangle \sigma_7) \rightharpoondown \{ A : \exp(\{ \} \alpha_7, \langle \rangle \sigma_7)\}\) exactly

\(a_2 \in \{ \} \alpha_6 \times \langle \rangle \sigma_6 \rightharpoondown \{ A : \exp(\{ C : \iota_3 \text{var}\}, \langle \iota_3 : \text{int}\})\}\) exactly

The row and storage variables \(\alpha_6\) and \(\sigma_6\) of \(a_1\) are filled in with \(\{\}\) and \(\langle\rangle\) to form the \(\text{exp}\) type.

We now show the inference for the second declaration. The right-hand side of the assignment statement has the denotation:
\[ a_5 = E[\text{use } X + \text{use } C] = (a_3 * a_4) ; \text{add}_{\text{int}} \]

The type inference for variable lookups \( a_3 \) and \( a_4 \) is the same as that for \( a_1 \), and we obtain the following schemes:

\[ a_3 \in \{X : \tau_4 \text{var}\} \alpha_8 \times (\tau_4 : \text{int})\sigma_8 \rightarrow \text{int} \]
\[ a_4 \in \{C : \tau_5 \text{var}\} \alpha_9 \times (\tau_5 : \text{int})\sigma_9 \rightarrow \text{int} \]

Parallel composition of these actions gives:

\[ (a_4 * a_3) \in \{C : \tau_4 \text{var}, X : \tau_5 \text{var}\} \alpha_{10} \times (\tau_4 : \text{int}, \tau_5 : \text{int})\sigma_{10} \rightarrow \text{int} \times \text{int} \]

This action is composed sequentially with \( \text{add}_{\text{int}} \in \text{int} \times \text{int} \rightarrow \text{int} \) to give the scheme:

\[ a_5 \in \{C : \tau_4 \text{var}, X : \tau_5 \text{var}\} \alpha_{10} \times (\tau_4 : \text{int}, \tau_5 : \text{int})\sigma_{10} \rightarrow \text{int} \]

The body of procedure \( P \) has the following action semantics denotation:

\[ a_7 = C[X := \text{use } X + \text{use } C] = (a_6 * a_5) ; \text{update} \]

where \( a_6 = ((\text{forget}_{(D,I,D)} ; \text{find}_{\text{var}} X) * \text{forget}_{(D,I,D)}) \) has the following scheme:

\[ a_6 \in \{X : \tau_6 \text{var}\} \alpha_{11} \times (\tau_6 : \text{var})\sigma_{11} \rightarrow \tau_6 \text{var} \times \tau_6 \text{var} \]

Parallel composition of \( a_6 \) and \( a_5 \) produces the scheme:

\[ (a_6 * a_5) \in \{C : \tau_4 \text{var}, X : \tau_7 \text{var}\} \alpha_{12} \times (\tau_4 : \text{int}, \tau_7 : \text{int})\sigma_{12} \]
\[ \rightarrow \tau_7 \text{var} \times \text{int} \times (\tau_4 : \text{int}, \tau_7 : \text{int})\sigma_{12} \]

which is sequentially composed with

\[ \text{update} \in (\tau_6 \text{var} \times \text{int}) \times (\tau_8 : \text{int})\sigma_{13} \rightarrow (\tau_8 : \text{int})\sigma_{13} \]
to give the substitution \([\iota_7 \mapsto \iota_9][\iota_8 \mapsto \iota_9][\sigma_{13} \mapsto \langle \iota_4 : \text{int} \rangle \sigma_{14}][\sigma_{12} \mapsto (\_ \sigma_{14})] \), and the following scheme for the procedure body:

\[
a_7 \in \{C : \iota_4 \text{var}, X : \iota_9 \text{var}\}_{\alpha_{12} \times \langle \iota_4 : \text{int}, \iota_9 : \text{int} \rangle \sigma_{14}} \mapsto \langle \iota_4 : \text{int}, \iota_9 : \text{int} \rangle \sigma_{14}
\]

The procedure declaration has the following denotation:

\[
a_8 = \mathcal{D}[\text{proc } P = X := \text{use } X + \text{use } C] = (\text{forget}_{(D, I, D)} ; \text{declare } a_7) ; \text{bind}_{cmd} P
\]

The type inference for \text{declare } a_7 fills in the storage variable \(\sigma_{14}\) of \(a_7\)'s scheme with \((\_ \sigma_{14})\) to obtain the scheme:

\[
(\text{declare } a_7) \in \{C : \iota_4 \text{var}, X : \iota_9 \text{var}\}_{\alpha_{12} \times (\_ \sigma_{15}) \mapsto \{P : cmd(\langle \iota_4 : \text{int}, \iota_9 : \text{int} \rangle)\}}_{\text{exactly}}
\]

The declaration has the typing scheme:

\[
a_8 \in \{C : \iota_4 \text{var}, X : \iota_9 \text{var}\}_{\alpha_{13} \times (\_ \sigma_{15}) \mapsto \{P : cmd(\langle \iota_4 : \text{int}, \iota_9 : \text{int} \rangle)\}}_{\text{exactly}}
\]

The scheme for the declaration block

\[
a_9 = \mathcal{D}[\text{macro } A = C \text{ and proc } P = X := X + C] = a_2 \star a_8
\]

unifies the source schemes and merges the target bindings schemes produced by \(a_2\) and \(a_8\) to give the scheme:

\[
a_9 \in \{C : \iota_4 \text{var}, X : \iota_9 \text{var}\}_{\alpha_{14} \times (\_ \sigma_{16}) \mapsto \{A : exp(\{C : \iota_3 \text{var}\}, \langle \iota_3 : \text{int} \rangle), P : cmd(\langle \iota_4 : \text{int}, \iota_9 : \text{int} \rangle)\}}_{\text{exactly}}
\]

We now show the type inference for the inner command block. The right-hand side of the assignment statement has the denotation:
\[ a_{10} = E[\text{expand } A] = ((\text{forget}_{DL,DI}) ; \text{find}_{exp} A) * \text{forget}_{DL,DI}) ; \text{eval} \]

The left term has kinding \( DL \vdash FDI \) and the scheme:

\[
\{A : \text{exp} \{ }\alpha_{16}, ( )\sigma_{17})\alpha_{17} \times ( )\sigma_{18} \\
\vdash \text{exp} \{ }\alpha_{16}, ( )\sigma_{17}) \times \{A : \text{exp} \{ }\alpha_{16}, ( )\sigma_{17})\alpha_{17} \times ( )\sigma_{18}
\]

The action \( \text{forget}_{DL,DI} \) passes along the bindings and storage arguments which are tupled with the \( \text{exp} \)-type value on the functional facet. \( \text{eval} \) has the typing scheme:

\[
eval \in \text{exp} \{ }\alpha_{18}, ( )\sigma_{19}) \times \{ }\alpha_{19} \times ( )\sigma_{20} \vdash \text{int} \\
\text{if } \{ }\alpha_{19} \leq \{ }\alpha_{18} \text{ and } ( )\sigma_{20} \leq ( )\sigma_{19}
\]

The type inference for \( ; \) unifies \( \{A : \text{exp} \{ }\alpha_{16}, ( )\sigma_{17})\alpha_{17} \) with \( \{ }\alpha_{19} \), and \( \{ }\alpha_{18} \) with \( \{ }\alpha_{18} \), and we get a new constraint on the declarative facet in the scheme for \( a_{10} \) that looks recursive:

\[
a_{10} \in \{A : \text{exp} \{ }\alpha_{20}, ( )\sigma_{21})\alpha_{21} \times ( )\sigma_{22} \vdash \text{int} \\
\text{if } \{A : \text{exp} \{ }\alpha_{20}, ( )\sigma_{21})\alpha_{21} \leq \{ }\alpha_{20} \text{ and } ( )\sigma_{22} \leq ( )\sigma_{21}
\]

The decomposition algorithm notes that both constraints have M-forms and no simplification is done. By Proposition 4.3, these constraints are trivially satisfied by instantiating both row variables to \( \{ } \) and both storage variables to \( ( ) \).

The assignment statement has the following denotation:

\[
a_{12} = C[C := \text{expand } A] = (a_{11} \ast a_{10}) ; \text{update}
\]

where \( a_{11} = ((\text{forget}_{DL,DI}) ; \text{find}_{\text{var}} C) * \text{forget}_{DL,DI}) \) has the scheme:

\[
a_{11} \in \{C : \nu_{11}\text{var}\}\alpha_{23} \times ( )\sigma_{23} \vdash \nu_{11}\text{var} \times ( )\sigma_{23}
\]
Parallel composition of $a_{11}$ and $a_{10}$ produces the scheme:

$$\{A : \exp(\{ \alpha_{20}, \langle \sigma_{21} \rangle \}, C : \iota_{11} \text{var} \alpha_{24} \times \langle \sigma_{24} \sigma_{24} \rangle \overset{\sigma_{24} \sigma_{24}}{\sim} (\iota_{11} \text{var} \alpha_{24} \times \text{int}) \times \langle \sigma_{24} \rangle)\}
$$

if $\{A : \exp(\{ \alpha_{20}, \langle \sigma_{21} \rangle \}, C : \iota_{11} \text{var} \alpha_{24} \leq \{ \alpha_{20}\}$

and $\langle \sigma_{24} \rangle \leq \langle \sigma_{21} \rangle$

Again, these constraints have M-forms and are satisfiable by Proposition 4.3. This scheme is composed sequentially with:

$$\text{update} \in (\iota_{12} \text{var} \times \text{int}) \times (\iota_{12} : \text{int}) \sigma_{25} \overset{\sigma_{25}}{\sim} (\iota_{12} : \text{int}) \sigma_{25}$$

and we obtain scheme

$$a_{12} \in \{A : \exp(\{ \alpha_{20}, \langle \sigma_{21} \rangle \}, C : \iota_{13} \text{var} \alpha_{24} \times (\iota_{13} : \text{int}) \sigma_{26} \overset{\sigma_{26}}{\sim} (\iota_{13} : \text{int}) \sigma_{26} \}
$$

if $\{A : \exp(\{ \alpha_{20}, \langle \sigma_{21} \rangle \}, C : \iota_{13} \text{var} \alpha_{24} \leq \{ \alpha_{20}\}$

and $(\iota_{13} : \text{int}) \sigma_{26} \leq \langle \sigma_{21} \rangle$

The second command of the inner block is the procedure call to $P$, whose denotation is:

$$a_{13} = C[\text{call } P] = ((\text{forget}_{DL,D}) \cdot \text{find } \text{cmd} P) \ast (\text{forget}_{DL,D}) ; \text{call}$$

The left-hand action has scheme:

$$\{P : \text{cmd}(\langle \sigma_{27} \rangle) \alpha_{25} \times \langle \sigma_{28} \rangle \overset{\sigma_{28}}{\sim} \text{cmd}(\langle \sigma_{27} \rangle) \times \langle \sigma_{28} \rangle$$

Action $\text{call}$ has scheme:

$$\text{call} \in \text{cmd}(\langle \sigma_{29} \rangle) \times \langle \sigma_{30} \rangle \overset{\sigma_{30}}{\sim} \langle \sigma_{30} \rangle \text{ if } \langle \sigma_{30} \rangle \leq \langle \sigma_{29} \rangle$$

The two actions are now sequentially composed and we have the following scheme for the procedure call:
The command sequence has denotation:

\[ a_{14} = C \left[ C := \text{expand } A ; \text{call } P \right] = (a_{12} * \text{forget}_{(D_I,D)}) ; a_{13} \]

The first command is composed with \( \text{forget}_{(D_I,D)} \) to pass along the bindings it receives to the second command. Next, this action is sequentially composed with \( a_{13} \). The type inference unifies the record \( \{A : \exp(\{\}a_{20}, (\sigma_{21}), C : \iota_{13}\text{var}\}a_{24} \) with the record \( \{P : \cmd((\sigma_{31}))a_{25} \) and the storage shape \( (\iota_{13} : \text{int})\sigma_{26} \) with the storage shape \( (\sigma_{32}) \). We obtain the typing scheme:

\[
a_{14} \in \{A : \exp(\{\}a_{20}, (\sigma_{21}), C : \iota_{13}\text{var}, P : \cmd((\sigma_{31}))a_{25} \times (\iota_{13} : \text{int})\sigma_{33}
\]

\[
\quad \rightarrow (\iota_{13} : \text{int})\sigma_{33}
\]

\[
\quad \text{if } \{A : \exp(\{\}a_{20}, (\sigma_{21}), C : \iota_{13}\text{var}, P : \cmd((\sigma_{31}))a_{25} \leq \{\}a_{20}
\]

\[
\quad \text{and } (\iota_{13} : \text{int})\sigma_{33} \leq (\sigma_{32} \text{ and } (\iota_{13} : \text{int})\sigma_{33} \leq (\sigma_{31})
\]

The constraint set is satisfiable as all constraints have M-forms.

The allocation of variable \( C \) has denotation \( a_{16} = \mathcal{M}[\text{var } C] = a_{16} ; a_{17}, \)

where

\[ a_{16} = (\text{forget}_{(D_I,I)} ; \text{allocint}) * \text{forget}_{(D_I,D)} \]

\[ a_{17} = (\text{forget}_{(FD_I,FD)} ; \triangleright (a_{15})) * \text{forget}_{(FD_I,I)} \]

where \( a_{15} = \text{forget}_{(FD,F)} ; \text{bind var } C \) has the typing scheme:

\[ a_{15} \in \iota_{14}\text{var} \times \{\}a_{27} \rightarrow \{C : \iota_{14}\text{var}\} \text{ exactly} \]

The type inference for \( \triangleright (a_{15}) \) extends \( a_{27} \) with a \( \Delta \)-field for \( C \), which the \( \iota_{14}\text{var} \) field in the target overrides. The other bindings in the source are allowed to flow through. We obtain the following typing scheme:
Actions $a_{16}$ and $a_{17}$ have the following schemes:

$$
\begin{align*}
&a_{16} \in \{\ell_1 : \text{undef}\} \sigma_{34} \vdash \ell_1 \text{var} \times \{\ell_1 : \text{int}\} \sigma_{34} \\
&a_{17} \in \iota_{14} \text{var} \times \{C : \Delta_4\} \alpha_{28} \times \{C : \iota_{44} \text{var}\} \alpha_{28} \times \{C : \iota_{44} \text{var}\} \alpha_{28} \times \{C : \iota_{44} \text{var}\} \alpha_{28} \times \{C : \iota_{44} \text{var}\} \alpha_{28}
\end{align*}
$$

The two actions are now sequentially composed and we obtain the following typing scheme for $a_{18}$:

$$
\begin{align*}
a_{18} \in \{C : \Delta_4\} \alpha_{30} \times \{\ell_1 : \text{undef}\} \sigma_{36} \vdash \{C : \ell_1 \text{var}\} \alpha_{30} \times \{\ell_1 : \text{int}\} \sigma_{36}
\end{align*}
$$

We now show the type inference for:

$$
\begin{align*}
a_{19} = C[\text{alloc var C in C := expand A; call P free}] = a_{18} \text{ in } a_{14}
\end{align*}
$$

The type inference unifies the target of $a_{18}$ with the source of $a_{14}$ to produce the substitution $[\ell_{13} \leftarrow \ell_1][\alpha_{30} \leftarrow \{A : \exp(\{\alpha_{20}, (\sigma_{21})\}, P : \cmd((\sigma_{31})\alpha_{31})[\alpha_{26} \leftarrow \{\alpha_{31}][\sigma_{30} \leftarrow \{\sigma_{37}][\sigma_{33} \leftarrow \{\sigma_{37}].$ The equivalence of the source and target imperative facet of $a_{14}$ is checked and we obtain the following typing scheme:

$$
\begin{align*}
a_{19} \in \{C : \Delta_4, A : \exp(\{\alpha_{20}, (\sigma_{21})\}, P : \cmd((\sigma_{31})\alpha_{31})\alpha_{31} \times \{\ell_1 : \text{undef}\} \sigma_{37}
\end{align*}
$$

The first call to procedure $P$ is composed in parallel with $\text{forget}_{(D_1,D)}$ when the commands are sequenced. $a_{20} = C[\text{call } P] \ast \text{forget}_{(D_1,D)}$ has the scheme:

$$
\begin{align*}
a_{20} \in \{P : \cmd((\sigma_{38})\alpha_{32} \times (\sigma_{39}) \vdash \{P : \cmd((\sigma_{38})\alpha_{32} \times (\sigma_{39})
\end{align*}
$$

if $(\sigma_{39} \leq (\sigma_{38})$
Next, $a_{20}$ is sequentially composed with $a_{19}$ and we obtain the scheme:

$$(a_{20} ; a_{19}) \in \{C: \Delta_4, A: \text{exp} \{\emptyset a_{20}, ()a_{21}\}, P: \text{cmd} \{()a_{40}\}\}a_{33}$$

$$\times (\ell_1: \text{undef})a_{41} \vdash (\ell_1: \text{undef})a_{41}$$

$$\text{if } \{A: \text{exp} \{\emptyset a_{20}, ()a_{21}\}, C: \ell_1 \text{var}, P: \text{cmd} \{()a_{40}\}\}a_{33} \leq \{\emptyset a_{20}\}$$

and $(\ell_1: \text{int})a_{41} \leq ()a_{21}$ and $(\ell_1: \text{int})a_{41} \leq ()a_{40}$

The type inference now overrides the declaration block $a_9$. The source record $$\{C: \ell_4 \text{var}, X: \ell_9 \text{var}\}a_{14}$$ is extended with fields for $A$ and $P$, producing the substitution $[a_{14} \leftarrow \{A: \Delta_5, P: \Delta_6\}a_{34}]$. The target record $$\{A: \text{exp} \{C: \ell_3 \text{var}\}, (\ell_3: \text{int})\}, P: \text{cmd} \{\ell_4: \text{int}, \ell_9: \text{int}\}\}a_{34} \text{ exactly is extended with "absent" fields for } C \text{ and } X.$$ The $A$ and $P$ fields in the target override the same fields in the source, and the $C$ and $X$ fields in the source, along with any other fields in $a_{34}$, are carried through to the target. We obtain the scheme:

$$\{C: \ell_4 \text{var}, X: \ell_9 \text{var}, A: \Delta_5, P: \Delta_6\}a_{34} \times ()a_{42}$$

$$\vdash \{A: \text{exp} \{C: \ell_3 \text{var}\}, (\ell_3: \text{int})\}, P: \text{cmd} \{\ell_4: \text{int}, \ell_9: \text{int}\}, \quad C: \ell_4 \text{var}, X: \ell_9 \text{var}\}a_{34} \times ()a_{42}$$

The above scheme actually shows the action composed in parallel with \texttt{forget}$(D_{II,I})$ to pass along the storage argument it receives to the command block.

The final step of the type inference composes this action sequentially with $a_{20} ; a_{19}$. We obtain the substitution $[a_{33} \leftarrow \{X: \Delta_7\}a_{35}][\Delta_7 \leftarrow \ell_9 \text{var}][a_{34} \leftarrow \{\}a_{35}][\Delta_4 \leftarrow \ell_4 \text{var}][a_{20} \leftarrow \{C: \ell_3 \text{var}\}][a_{21} \leftarrow (\ell_3: \text{int})][a_{40} \leftarrow (\ell_4: \text{int}, \ell_9: \text{int})][a_{41} \leftarrow (\ell_9: \text{int})][a_{42} \leftarrow (\ell_1: \text{undef})]a_{43}$, and the following typing scheme for the program:
We may simplify this scheme by eliminating the constraints. First note that \( \ell_3 \) must equal \( \ell_1 \) for the record constraint to hold true. Once this substitution is made, the record constraint is true for any instantiation of \( \alpha_{35} \). The first constraint on the imperative facet now holds for any instantiation of \( \sigma_{43} \). The remaining constraints require that \( \sigma_{43} \) be extended with \( \ell_4 \) and \( \ell_9 \) cells. Applying the substitution \( U = [\ell_3 \mapsto \ell_1][\sigma_{43} \mapsto (\ell_4: \text{int}, \ell_9: \text{int})\sigma_{44}] \) to the constraints, we obtain

\[
\{A: \text{exp}(\{C: \ell_1 \text{var}\}, (\ell_1: \text{int})), P: \text{cmd}(\langle \ell_4: \text{int}, \ell_9: \text{int} \rangle), \\
X: \ell_9 \text{var}, C: \ell_1 \text{var}\} \alpha_{35} \leq \{C: \ell_1 \text{var}\} \\
and (\ell_1: \text{int})\sigma_{43} \leq (\ell_3: \text{int}) \quad \text{and} \quad (\ell_1: \text{int})\sigma_{43} \leq (\ell_4: \text{int}, \ell_9: \text{int}) \\
and (\ell_1: \text{undef})\sigma_{43} \leq (\ell_4: \text{int}, \ell_9: \text{int})
\]

These constraints simplify to M-forms which are trivially satisfiable. We omit them from the scheme, giving us the final result:

\[
\{C: \ell_4 \text{var}, X: \ell_9 \text{var}, A: \Delta_5, P: \Delta_6\} \alpha_{35} \times (\ell_1: \text{undef}, \ell_4: \text{int}, \ell_9: \text{int})\sigma_{43} \\
\rightarrow (\ell_1: \text{undef}, \ell_4: \text{int}, \ell_9: \text{int})\sigma_{43}
\]

The field variables \( \Delta_5 \) and \( \Delta_6 \) in the scheme's source are "little row variables"; since neither they nor the row variable \( \alpha_{35} \) appear in the scheme's target, we compress
them into a single row variable $\beta$ and obtain the scheme:

$$\{C: t_4\text{var}, X: t_9\text{var}\} \times (\ell_1: \text{undef}, t_4: \text{int}, t_9: \text{int}) \sigma \rightarrow (\ell_1: \text{undef}, t_4: \text{int}, t_9: \text{int}) \sigma$$

for the program.

The scheme tells us that the program requires a declarative facet argument that has bindings of $C$ and $X$ to variables and may contain other bindings as well. The program also requires an imperative facet argument with cells for the two variables. Note that the exact locations of the two memory cells are "unknown"; they are both $\ell$-variables. The storage may contain other cells, but the location $\ell_1$ must be unallocated. Recall that locations are generated by the function $\text{newlocn}$, so the location $\ell_1$ is, in fact, arbitrary. We have not studied the precise nature of storage cell location generation and its implications in action semantics.

The scheme illustrates that the typing annotations for an action semantics denotation contain more information than those for the lambda-calculus style denotational semantics. Rather than simply saying the program's denotation is a function with domain $\text{Environment} \rightarrow \text{Store} \rightarrow \text{Store}$, we know that its "environment" must contain bindings for $C$ and $X$ to variables, and that its "store" must contain cells for these variables. The scheme also expresses the polymorphic nature of the action's denotation. The program can be used in any declarative-imperative context that has at least bindings for $C$ and $X$ and holds their storage cells. The scheme shows us that the program does not change the shape of the storage.

We also have some intuitive indication of what the program does from its typing scheme: we know that it examines and possibly modifies the memory cells for $C$ and $X$. By the soundness and completeness properties of the type schemes, the scheme describes all the well-defined behaviors of the program.
Implementation of the Type Inference Algorithms

We have seen in the previous section that annotating an action semantics expression with type schemes is a tedious task. In this section we describe our implementation of the type inference algorithm in ML. We include a small bit of the ML code.

Figure 5.9 lists the ML datatypes that implement the type schemes in each facet. The code is straightforward from the syntax of the type schemes given in Chapter 3. Our implementation includes records in the functional facet for experimentation with language definitions that operate upon records. The type scheme \( \{ A: \text{present}(\text{int}), B: \text{absent}\} \alpha_{\text{exactly}} \) is implemented as:

\[
\text{bindings(} \begin{array}{ll}
A & \text{present(prim(Int))} \\
B & \text{absent}
\end{array}, \alpha_{\text{ex('bl')}} \text{)}
\]

where the string "'bl" is a bindings (row) type variable.

The ML datatypes that implement the type schemes for actions are given in Figure 5.10. The ML type pattern serves two purposes: it tags a type scheme with the facet it belongs to (e.g., Bindings), and it is used in defining substitutions:

\[
\text{typedef subst} = \\
\text{failure | subst of TypeVar list * (TypeVar -> pattern)}
\]

The constructor tag on the pattern matches the form of type variable it is bound to in the substitution.

The source and target of an action are implemented as pairs, where the first component (kinds) specifies the facets the source or target is defined on, and the
datatype Prim = Nat | Int | Real | Bool

datatype locn = iota of TypeVar | locn of int

datatype Values = prim of Prim | pair of Values * Values |
                 var of locn * Values | fcn of Values * Values |
                 theta of TypeVar | record of Bindings

and Bindings = bindings of (string * Field) list * Row

and Row = ex | alpha of TypeVar | alpha' ex of TypeVar

and Field = absent | delta of TypeVar | present of Values

and Storage = cells of (locn * cell) list * Top

and cell = undef | full of Values

and Top = empty | gamma of TypeVar

Figure 5.9. ML Implementation of Facets
datatype pattern = Values of Values | Field of Field | Bindings of Bindings | Storage of Storage | Loc of locn | ns

datatype kinds = one | F | D | I | FD | FI | DI | FDI

datatype facet_designator = Ff | Df | Sf

datatype constraint = vxv of Values * Values |
  sxs of Storage * Storage |
  False

datatype scheme = ppc of (kinds * (facet_designator -> pattern)) * |
  (kinds * (facet_designator -> pattern)) * |
  constraint list |
  ss of scheme * scheme |
  failed

Figure 5.10. ML Implementation of Schemes

second component (facet_designator -> pattern) is a function that maps each facet designator to its type scheme. Action schemes have three forms: the constructor ppc tags a scheme of the form $S_1 \rightarrow S_2$ if $C$; the constructor ss tags a scheme of the form $S_1/S_2$, and the constructor failed represents the scheme $ns \rightarrow ns$ if false. An action scheme's constraint set is represented as a list, where the empty list represents true. For example, the scheme for the action $\text{succ} \in \theta \rightarrow \theta$ if $\theta \leq \text{real}$ is implemented as:

$$ppc((F, fn d => \text{if } d = Ff \text{ then } \text{Values(theta("'t'i")) else ns}) ,$$

$$(F, fn d => \text{if } d = Ff \text{ then } \text{Values(theta("'t'i")) else ns}) ,$$

$$[\text{vxv(theta("'t'i"), prim(Real))}]$$

Facets which are not in an action's source or target kinds component map to the
fun Analyze (A: Action): AnnotatedAction * scheme = case A of
    find Ide =>
        let val Theta = NewValuesVar();
        val Alpha = alpha(NewRowVar());
        val S = Bindings(bindings([(Ide, present Theta)], Alpha);
        val T = Values(Theta);
        val sch = ppc ( (D , (fn d => if d = Df then S else ns)),
                    (F , (fn d => if d = Ff then T else ns)),
                    [] )
        in ( Find(Ide, sch), sch) end |
    bind Ide =>
        let val Theta = NewValuesVar();
        val S = Values(Theta);
        val T = Bindings(bindings([(Ide, present Theta)], ex));
        val sch = ppc ( (F , (fn d => if d = Ff then S else ns)),
                      (D , (fn d => if d = Df then T else ns)),
                      [] )
        in ( Bind(Ide, sch), sch) end |
...

Figure 5.11. ML Implementation of find i and bind i

pattern ns in its (facet_designator -> pattern) component.

The datatype Action implements the syntax of actions:

datatype Action =
    forget of kinds * kinds |
    find of string | bind of string | ...
    semi of Action * Action | ...

Each action combinator is an ML type constructor on the datatypes of its component actions. For example, the action find A ; bind B is implemented as the ML term semi(find "A", bind "B").

The type inference algorithm is implemented as a function called Analyze that
maps an Action to a pair consisting of an annotated syntax tree for the action, and a typing scheme. (See Figure 5.11.) The datatype AnnotatedAction pairs each subaction in an action semantics expression with its typing scheme:

```
datatype AnnotatedAction =
    Forget of kinds * kinds * scheme |
    Find of string * scheme | Bind of string * scheme | ...
    Semi of AnnotatedAction * AnnotatedAction * scheme | ...
```

Analyze is implemented as a cases statement on the datatype Action. For the primitive actions, the function takes care of allocating fresh type variables for each action. ML code for the type inference for find i and bind i is given in Figure 5.11. The code is straightforward from their schemes in Figure 3.2.

For the action combinators, Analyze first infers types for each component action and then combines their schemes. Sequential composition is implemented as:

```
semi (A1,A2) => let val (aal,Tl) = Analyze A1;
              val (aa2,T2) = Analyze A2;
              val sch = dropfail (semitype (T1,T2))
              in ( Semi(aal,aa2,sch) , sch ) end
```

Function semitype does the type inference on the schemes T1 and T2, and function dropfail eliminates failed schemes from a slash scheme.

semitype is defined in Figure 5.12. For two ppc schemes, the construction of the result scheme is that presented in Chapter 3. Function unify operates on two (kinds * (facet_designator -> pattern)) pairs: first, equivalence of kinds is checked; then the patterns of each facet are unified. Function prunion applies a substitution
fun semitype ( ppc(p1,p2,c) , T2 ) = (case T2 of
  failed => failed |
  ss(s1,s2) => ss( semitype( ppc(p1,p2,c) ,s1) ,
                  semitype( ppc(p1,p2,c) ,s2) ) |
  ppc(p1',p2',c') =>
    (let val U = unify (p2,p1',EmptySubst);
     val d = prunion U (c ® c') in
     if (isfailure U) orelse not(issat d) then failed
     else let val q1 = PruneScheme U p1
        and q2 = PruneScheme U p2'
        in ppc( q1 , q2 , d ) end
    end)) |
semitype ( ss(s1,s2) , T2 ) =
  ss( semitype(s1,T2) , semitype(s2,T2) ) |
semitype ( failed , T2 ) = failed

Figure 5.12. ML Implementation of Sequential Composition

to a list of constraints, removing any duplicates that result from its application.
Function Prunescheme applies a substitution to a source or target scheme.

Our ML implementation includes the decomposition and satisfaction algorithms
discussed in Chapter 4, and a pretty-printer that translates an action's ML typing
scheme into a string of characters that resembles its typing scheme given in Chapter 3.
We have used the implementation to annotate the action semantics denotations of
programs in a small imperative language and also a small expression language.
CHAPTER 6. CONCLUSION

Related Work

Our type inference algorithms for action semantics are based on the work of a number of researchers. In this section we discuss each of these different type systems and point out where our ideas originated. We also give some comparisons with our system where appropriate. We begin with a description of ML-style type inference, followed by various extensions of it to subtypes and records.

Milner-style type inference

Unification-based, polymorphic type inference originated with the programming language ML, which includes the lambda calculus as a subset. In [18], Milner defines a type inference algorithm for a subset of ML and proves its soundness with respect to a semantic model.

The type inference algorithm assigns type schemes (what Milner calls polytypes) to expressions in the language. For example, the term $\lambda x.x$ is given the type scheme $\alpha \to \alpha$ which is interpreted as the collection of types $\forall \alpha . \alpha \to \alpha$, where $\alpha$ ranges over all types. Each variable in a type scheme is quantified at the outermost scope. (Schemes such as $(\forall \alpha . \alpha \to \alpha) \to (\forall \alpha . \alpha \to \alpha)$ are not allowed.) A type environment gives types to the free identifiers in an expression. For example, when
analyzing the body $x + y$ of the expression $\lambda y . \lambda x . x + y$, the type environment would have form $\{x : \alpha, y : \beta\}$. Because action semantics is a combinator notation, our type inference algorithms differ from those based on the lambda calculus in that we do not need to maintain a type environment for identifiers. This simplifies our algorithms.

The main interest of Milner's type inference algorithm lies in its different treatment of the language constructs `\(\lambda i e\)` and `let \(i = e_1\) in \(e_2\)`. The semantics for the two constructs are equivalent (i.e., \([\lambda i e_2] e_1\] is equivalent to \[let i = e_1\) in \(e_2\])

but in some cases it is possible to assign a type to the let construct but not to the lambda construct. Lambda binding does not allow different occurrences of a \(\lambda\)-bound variable to have different types. For example, the term

\[(\lambda f . (f\ true, f\ 3)) (\lambda x . x)\]

is ill-typed because \(f\)'s domain scheme \(\alpha\) cannot be unified to both types `bool` and `int`. Although terms may have more than one type, with lambda binding there is no way to take advantage of this polymorphism.

(Parametric) polymorphism is made available in ML through the `let` binding mechanism. For example, the type inference for the term

\[let f = \lambda x . x in (f\ true, f\ 3)\]

gives the different applications of \(f\) schemes with fresh variables. Unification is done for each application, but the substitutions are not applied to the type scheme specified for \(f\) by its `let`-binding in the type environment. Milner explains that, intuitively, the difference between the two binding mechanisms exists because a \(\lambda\)-abstraction
may occur without an argument, whereas the let construct involves the restricted use of an abstract.

Subtyping

Mitchell  Parametric polymorphism can only represent "structural" inclusion among types (as described by Mitchell and Fuh and Mishra's relation MATCH discussed in Chapter 4), and not subtyping relations such as \texttt{int} \leq \texttt{real}. Mitchell \cite{20} extended a subset of ML (namely the lambda calculus) with subtypes. The model of subtypes was based on set containment: if \( \tau_1 \subseteq \tau_2 \), then \([\tau_1] \subseteq [\tau_2]\). Mitchell noted that in the presence of subtypes, an ML-style type scheme does not characterize all possible typings of each term. He therefore introduced coercion sets (constraint sets) to the type schemes. Mitchell explains coercions as hypotheses about the types of the free variables and the subtype relationships between these types. This new form of type scheme requires that substitutions into a scheme must preserve the satisfiability of its coercion set. Mitchell omitted the algorithms for checking satisfiability from \cite{20}.

Fuh and Mishra  Fuh and Mishra built on Mitchell's work by implementing the algorithms for checking the satisfiability of a constraint set \cite{10}. In \cite{11}, Fuh and Mishra noted that their type inference system generated coercion sets with size proportional to program size. The problem was that their type inference algorithm introduced coercions for every occurrence of an identifier and for every function application. For example, type inference for the term \((\lambda x . x) y\) would infer the type
schemes

\[ \lambda x . x \in \theta_1 \rightarrow \theta_2 \text{ if } \theta_1 \leq \theta_2 \quad \text{and} \quad y \in \theta_4 \text{ if } \theta_3 \leq \theta_4 \]

(with the type environment \( \{ y : \theta_3 \} \)) and generate the type scheme

\[ \theta_2 \text{ if } \{ \theta_1 \leq \theta_2, \theta_3 \leq \theta_4, \theta_4 \leq \theta_1 \} \]

for the application. Large constraint sets make type schemes difficult for the user to decipher and also affect the efficiency of the type inference algorithm.

**Action semantics** The model for action semantics discussed in Chapter 2 gives us an advantage for type inference: it reduces the size of constraint sets. Because the typing functions are monotonic, for any action \( a \) and any type \( t \), if \( T_a(t) \neq \text{ns} \), then for all types \( t' \leq t \), \( T_a(t') \neq \text{ns} \). This means that action \( a \) is defined on all types \( t' \) which are subtypes of \( t \), and a coercion is not needed for the application of action \( a \) to a value of type \( t' \). Further, action composition unions the constraint sets of the component actions: no new constraints are introduced into the type scheme for the composed action. Constraints are only needed to restrict the types a variable in a scheme may be instantiated to (e.g., \( \text{succ}, \text{add}_{\text{int}}, \text{and} \ add_{\text{real}} \)), or to express relations between the variables in a scheme (e.g., \( \text{update}, \text{eval}, \text{and} \ \text{call} \)).

**Records**

**Cardelli** Cardelli studied the notions of record type and inclusion subtyping among record types in his typing system for a language with multiple inheritance [3]. One record type is a subtype of another if it contains all of its fields. For example, \( \{ a : \text{int}, b : \text{bool} \} \) is a subtype of \( \{ a : \text{int} \} \). Inclusion subtyping corresponds to the
ordering on our non-exactly record types given in Chapter 2. Cardelli’s system does not use type schemes to express polymorphism, instead all terms are explicitly typed and the subtype relation on types is checked.

Cardelli and Wegner Cardelli and Wegner study the use of explicit universal quantification to model parametric polymorphism, and explicitly bounded quantification of types to model subtypes and multiple inheritance [5]. For example, the type \( \forall \alpha \leq \{A:\text{int}\} . \alpha \rightarrow \text{int} \) is the type of a function that accepts any record having at least an integer \( A \) component and extracts its contents. Bounded quantification allows us to express input-output dependencies in a type scheme. For example, the result type of a function with type \( \forall \alpha \leq \{A:\text{int}\} . \alpha \rightarrow \alpha \) will be the same as its argument type, whatever subtype of \( \{A:\text{int}\} \) it happens to be. This was not possible in Cardelli’s earlier system [3].

Wand Wand studied type inference for a language based on Cardelli’s system [41]. Wand introduced row variables to provide a mechanism for multiple inheritance. Row variables allow type inference on records to be reduced to unification so that a constraint set is not needed to express subtype relations. For example, the function \( \text{succA} = \lambda r . (r.A) + 1 \) has the scheme \( \{A:\text{int}\} \alpha \rightarrow \text{int} \). (\text{succA} corresponds to the action \textit{find A; succ}.) It may be applied to records with fields other than \( A \) such as \( \{A = 3; B = \text{true}\} \) because the argument’s type scheme \( \{A:\text{int}, B:\text{bool}\} \) unifies with the function’s domain scheme \( \{A:\text{int}\} \alpha \). In Cardelli’s system, \( \text{succA} \) is given the typing \( \{A:\text{int}\} \rightarrow \text{int} \) and may be applied to the same record, since \( \{A:\text{int}, B:\text{bool}\} \leq \{A:\text{int}\} \).

Wand’s language includes a “with” construct that allows an arbitrary record to
be extended with a new field or the value in an existing field to be changed. The
with construct allows the definition of an "update" function such as

\[ \text{incdate} \cdot A = \lambda r \cdot r \textbf{with } A := (r.A) + 1 \]

Wand's type inference algorithm would assign \( \text{incdate} \cdot A \) the scheme \( \{A: \text{int}\} \alpha \rightarrow \{A: \text{int}\} \alpha \), which expresses the bounded polymorphism \( \forall \alpha \leq \{A: \text{int}\} . \alpha \rightarrow \alpha \) of Cardelli and Wegner's system. \( \text{incdate} \cdot A \) corresponds to the action \( \triangleright (\text{succ} \cdot A \cdot \text{bind} A) \), which, for the actions in Chapter 5, would be given the same scheme.

Wand points out that the term

\[ \text{letfix} = (\lambda f \cdot f \{A = 3; B = 5\} + f \{A = 3; C = 7\}) \text{succ} \cdot A \]

is not well-typed in his system: the row variable \( \alpha \) in the scheme for \( \text{succ} \cdot A \) must be instantiated differently for each occurrence of the identifier \( f \) to which it is bound. This term is, however, well-typed in Cardelli's system.

**Action semantics** We have a similar problem typing a term such as \( \text{letfix} \) in action semantics. If we consider only the scheme for \( \text{succ} \cdot A \) as being bound to \( f \) in the declarative facet, our problem seems to be exactly the same as Wand's. But the semantics of \( \text{letfix} \) clarifies the problem. When an action is to be abstracted and placed on the functional facet, a single morphism of the action must be selected (i.e., the action must be made monomorphic). Hence, an abstracted action can no longer be applied to values with different types. In Chapter 5, we saw a way to recover the polymorphism of an abstract with the actions \( \text{call} \) and \( \text{eval} \). In the previous example, the identifier \( f \) could be bound to the action expression abstract \( \text{define}(\text{succ} \cdot A) \in \{A: \text{int}\} \rightarrow \text{int} \), and enacted in the different contexts where \( f \) is
applied by the action `find f; eval`. This solution was at the expense of removing subtyping on non-record types from the functional facet, however.

**Stansifer**  Stansifer studied type inference for a language with records and inclusion subtyping on records similar to Cardelli's [3]. Stansifer introduced constraints on records types. For example, he would give `succA` the typing scheme

\[ \alpha \rightarrow \text{int if } \alpha \leq \{A:\text{int}\} \]

Stansifer's system is capable of assigning the following type scheme to `letfix`:

\[
\text{int if } \alpha \leq \{A:\text{int}\} \text{ and } \beta_1 \leq \{A:\text{int}, B:\text{int}\} \text{ and } \beta_1 \leq \alpha \\
\text{ and } \beta_2 \leq \{A:\text{int}, C:\text{int}\} \text{ and } \beta_2 \leq \alpha
\]

However, Stansifer does not give an algorithm for checking the satisfiability of these constraints. He merely points out that "one must know whether a solution exists or not before one can conclude an expression has a type." His constraints are similar to our constraints on non-exactly records discussed in Chapter 4, so he could probably use our decomposition algorithms.

**Jategaonkar and Mitchell**  Jategaonkar and Mitchell [15] combine Mitchell's system [20] with that of Wand [41] in an extension of ML called ML++. The language features a binding mechanism for row variables in its syntax called extended pattern matching. This allows one to define a function such as

\[ incA = \lambda \{a = x; v\} . \{a = x + 1; v\} \]

which updates a record. The identifier `v` is bound to the fields other than `a` in an actual argument record. `incA` is assigned the type scheme \( \{a:\text{int}, \alpha\} \rightarrow \{a:\text{int}, \alpha\} \)
and may be applied to a record such as \( \{ a = 1; b = \text{true} \} \) to obtain the result \( \{ a = 2; b = \text{true} \} \). In action semantics, this behavior is seen on storage records in the imperative facet.

Jategaonkar and Mitchell’s type inference algorithm keeps track of type expressions that appear during type inference but not in the final typing statement of a term. These type expressions are used to place restrictions on the allowable substitutions into row variables. For example, in the type inference for

\[
(\lambda \{ a = x; u \} . \{ u \}) (\{ a = 3; v \})
\]

the lambda term is given the scheme \( \{ a:\text{int}; \alpha \} \rightarrow \{ \alpha \} \). But (without restrictions) the final typing scheme \( \{ \alpha \} \) erroneously allows \( \alpha \) to be instantiated to \( \{ a:\text{bool}; \beta \} \) because the “context” \( \{ a:\text{int}; \alpha \} \) has been lost. The type expression \( \{ a:\text{int}; \alpha \} \) is called a cut formula because the scheme has been “cut” from the result typing scheme. Cut formulas are paired with type schemes to form a “restricted typing statement.” Cut formulas are generated by the type inference rule for function application.

Jategaonkar and Mitchell use a set of atomic constraints as in [20] that may contain atomic row constraints of the form \( r_1 \leq r_2 \), where \( r_1 \) and \( r_2 \) are either row variables or \( \{ \} \). Row constraints are generated at every occurrence of a record, just as subtyping constraints were generated at every occurrence of an identifier in the Mitchell/Fuh and Mitchell system. For example, the record \( \{ a = x; u \} \) generates the type environment \( \{ x : r_1, u : \alpha \} \), the type scheme \( \{ a : r_2; \beta \} \), and the constraint set \( \{ r_1 \leq r_2, \alpha \leq \beta \} \).

Substitutions into the constraint set, subject to the set of restrictions, may extend row variables with additional fields. Structural subtyping is used on records, hence, records having identical label sets may be decomposed to atomic relations.
The type of a record pattern always contains a row variable so that record functions may be applied to records with additional fields not referenced by extended pattern matching. But this also assists in the decomposition of record constraints.

Jategaonkar and Mitchell's algorithm seems complex to us, and they state that it is an open problem to develop a semantic model for ML⁺.

Remy  Remy introduced the notions of "present" and "absent" fields, and field variables as an alternative to Wand's row variables [31]. He studied a lambda calculus language with records. All records are defined over the same finite set of labels $L$, and instead of a row variable in the type scheme for a record, unknown fields are explicitly listed out. For example, given labels $A$, $B$, and $C$, the scheme $\{A:\text{int}\}_{\alpha}$ would be written as $\{A: \text{present(int)}, B: \Delta_B, C: \Delta_C\}$. An interesting capability made available by field variables is their ability to express relations between the fields in a record, regardless of whether the fields are absent or present. For example, a function $\text{swapAB}$ may be given the scheme $\{A: \Delta_A, B: \Delta_B, C: \Delta_C\} \rightarrow \{A: \Delta_B, B: \Delta_A, C: \Delta_C\}$.

Remy uses a structural notion of subtyping. He points out that when non-structural subtyping is extended to non-atomic types in his system, it is not known how to simplify (decompose) a constraint set. In particular, he mentions the constraint form $\text{present}(t) \leq \text{absent}$. Remy does not give a semantic model for his language.

Wand  Wand introduced a record concatenation operation to the lambda calculus with records [42]. He showed that in the presence of record concatenation, his language does not have principal types. Instead, every well-typed term has a finite set of type schemes such that every typing is an instance of one of these schemes.
Wand's system uses Remy's field variables and absent and present\( (t) \) fields in addition to row variables for record type schemes. Wand introduced the concept of "domain of use" for a row variable: instead of listing fields for all of the labels in the program, all records with the same row variable have the same explicit labels. Thus, only the labels with fields known to be present or absent need be explicitly listed in a record scheme. The domain of use supercedes Jategaonkar and Mitchell's use of cut formulas. This approach was used in our type inference algorithms in Chapter 3.

Wand models records as total functions \( L \to (V + \{\text{absent}\}) \), where \( L \) is a fixed set of labels, and \( V \) is a domain of values. His record concatenation operation is defined as

\[
\text{concat}(r_1, r_2) = \lambda i . \text{isV}(r_2(i)) \to r_2(i) \mid r_1(i)
\]

Present fields in \( r_2 \) override fields in \( r_1 \).

Type inference for \( \text{concat}(r_1, r_2) \) assigns schemes

\[
r_1 \in \{i : \Delta_i\} \alpha_1 \quad \text{and} \quad r_2 \in \{i : \Delta'_i\} \alpha_2
\]

to the argument records, where \( i \) ranges over the set of labels used in the program. \( \text{concat} \) is given the following scheme:

\[
\text{concat} \in \{i : \Delta_i\} \alpha_1 \times \{i : \Delta'_i\} \alpha_2 \to \{i : \Delta''_i\} \alpha_3 \quad \text{if} \quad \alpha_1 \parallel \alpha_2 = \alpha_3
\]

where \( \alpha_1 \parallel \alpha_2 = \alpha_3 \) is an extension constraint that abbreviates the set of disjunctions

\[
(\Delta'_i = \text{present}(t_i) \text{ and } \Delta''_i = \Delta'_i) \text{ or } (\Delta'_i = \text{absent} \text{ and } \Delta''_i = \Delta'_i)
\]

For each field \( i \), either the field is present in \( r_2 \) (i.e., \( \Delta'_i = \text{present}(t_i) \)), and thus present in the result, or else the field is absent in \( r_2 \), in which case the field in the result is the same as it is in \( r_1 \), whether it be present or absent.
SCHEME CONCEPT INTRODUCED BY

<table>
<thead>
<tr>
<th>Type variables</th>
<th>Milner [18]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint sets</td>
<td>Mitchell [20]</td>
</tr>
<tr>
<td>Row variables</td>
<td>Wand [41]</td>
</tr>
<tr>
<td>Field variables</td>
<td>Remy [31]</td>
</tr>
</tbody>
</table>

Figure 6.1. Advances in Type Schemes

**Action semantics** Complex constraints are not needed for record concatenation ("override") in action semantics because the typing schemes for actions have the \( \alpha \)- and \( \Delta \)-preserved properties discussed in Chapter 3. The properties ensure us that if the same identifier is bound to a field variable in both records, it is the same field variable (i.e., \( \Delta_i = \Delta_i' = \Delta_i'' \)), so a constraint is not needed. This represents the harmless case where an unknown binding is overriding itself.

**Ongoing and Future Research**

Actions have more detailed typing annotations than those usually found in denotational semantics definitions. For example, in addition to type annotations related to the values that flow through semantics equations, there are annotations for sets of bindings that should be present in an environment. In the future, we plan to study how typing information such as this can indicate language structure. For example, an action semantics might prove useful in detecting important language properties or concepts such as whether a language is dynamically or statically scoped, sequential imperative, statically typed, or block-structured (in its store; in its environment bindings). We plan to investigate the following topics, among others.
Non-interference  We plan to use the set of cells model of storage presented here to study non-interference properties of an action semantics expression [34]. For example, actions in the parallel composition $a_1 \ast a_2$ may operate independently without need for mutual exclusion when they operate on non-interfering sets of storage cells. The typing annotations on the imperative facet indicate which cells are used by an action.

Stack-based storage  We also plan to investigate imperative facet types that are stacks of cells [33]. This formulation will hopefully prove useful in expressing the semantics of block-structured languages in action semantics. We have worked with primitive actions on the imperative facet that were parameterized on stack depth. In doing so, we avoided constraints on storage types for the primitives update and contents, but limited the usefulness of the actions. In general, this is information a user would not be able to supply. On the other hand, when the stack depth is unknown, it is unclear what forms of constraint are necessary for storage types, whether a simplifying concept similar to the row variable exists when the storage shape is stack-based, and whether there is an algorithm for checking satisfiability. We have used sets of storage cells as types in the imperative facet here mainly because of their similarity to record types in the declarative facet.

Analysis of semantics equations  The type inference algorithms described in this report are for a specific action semantics expression. We are currently working on a type inference algorithm designed to analyze the semantics equations themselves. An equation such as:

$$\mathcal{E}[E_1 + E_2] = (\mathcal{E}[E_1] \ast \mathcal{E}[E_2]) \; \text{add}_{\text{real}}$$
is dependent on the other equations in the syntax of expressions, and possibly other syntactic constructs as well. We do not know what facets flow into an equation by looking at the equation itself. For example, depending on whether identifiers may be bound to variables, the imperative facet may or may not be needed for expression evaluation.

**A choice or cases combinator**  In Figure 1.4, we used an action called `choose` to implement a conditional statement, however, we did not define the semantics for this action. We would like `choose` to be defined in terms of the conditional combinator `/`. In [8], we use the primitive types `tt` and `ff`, where `tt ≤ bool` and `ff ≤ bool`, to define the semantics for a conditional statement:

\[
C[\text{if } B \text{ then } C_1 \text{ else } C_2] = B[B][C_1] / (istt \; §, \; C[C_1]) / (isff \; §, \; C[C_2])
\]

This equation is used with "constant" boolean expressions, constructed with the BNF rule:

\[
B ::= \text{true} | \text{not } B | I
\]

If boolean expressions of the form \( E_1 = E_2 \) were added to the language, the above definition would no longer be correct: the result of \( B[B] \) would be of type `bool`, which is neither type `tt` nor type `ff`. Some sort of "lifting" operation is needed to convert the action \((istt \; §, \; C[C_1]) / (isff \; §, \; C[C_2])\) to an action on type `bool`. We would like to generalize this idea to a form of cases statement.

Future work also includes the study of recursive actions, and language-specific collections of data types such as denotable, expressible, and storable values.
BIBLIOGRAPHY


