A TRUE HIGHER-ORDER MODULE SYSTEM

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
GEORGE KUAN

CHICAGO, ILLINOIS
JUNE 2010
To my dearest wife
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>ix</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>xi</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xi</td>
</tr>
</tbody>
</table>

## CHAPTER

1 INTRODUCTION .............................................................. 1
   1.1 Higher-Order Module System .................................... 2
   1.2 Full Transparency and True Separate Compilation ........... 7
   1.3 Organization .................................................... 8

2 BACKGROUND .............................................................. 10
   2.1 Module Systems in Other Languages ............................ 12
   2.2 The ML Module System .......................................... 13
   2.3 Separate Compilation .......................................... 15
   2.4 Formal Accounts of ML-based Module Systems ............... 16

3 THE DESIGN SPACE ....................................................... 34
   3.1 Applicative Versus Generative Functor Semantics .......... 34
   3.2 First- Versus Second-Class Modules .......................... 40
   3.3 Recursive Modules ............................................. 41
   3.4 Signature Ascription .......................................... 41
   3.5 Separate Compilation .......................................... 42

4 TYPE SYSTEM ............................................................. 46
   4.1 Kind System .................................................... 47
   4.2 Semantic Type System .......................................... 49

5 SURFACE LANGUAGE ....................................................... 53
   5.1 Core Language .................................................. 53
   5.2 Signatures ...................................................... 53
   5.3 Module Calculus ............................................... 55
   5.4 Primary and Secondary Components ............................ 55
   5.5 Nonvolatile and Volatile Type Constructors ................. 56

6 ENTITY CALCULUS ......................................................... 58
   6.1 Entity declarations ............................................ 61
   6.2 Notation ........................................................ 62
   6.3 Entity Evaluation Semantics ................................... 62
   6.4 Relativization .................................................. 66
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Apply functor example in OCaml</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Signature assigned to Apply functor under MacQueen-Tofte semantics</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Signature assigned to Apply functor under applicative functor semantics</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>OCaml and SML/NJ versions of ApplyToInt functor</td>
<td>4</td>
</tr>
<tr>
<td>1.5</td>
<td>Examples of applicative functors and true higher-order functors</td>
<td>5</td>
</tr>
<tr>
<td>1.6</td>
<td>SymbolTable functor example from Dreyer [13]</td>
<td>6</td>
</tr>
<tr>
<td>1.7</td>
<td>Set functor example from Dreyer [13]</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Sharing type constraint example</td>
<td>16</td>
</tr>
<tr>
<td>2.2</td>
<td>The spectrum of major module system families</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Biswas's elaboration rule for abstract type specifications: $\Gamma$ is the type environment mapping program variables to types. $W$ is a list of formal parameters variables the specification may depend on.</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>SML90 structure sharing example</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>SML90-style structure sharing</td>
<td>29</td>
</tr>
<tr>
<td>2.6</td>
<td>A comparison of major ML-like module systems</td>
<td>33</td>
</tr>
<tr>
<td>3.1</td>
<td>Set functor</td>
<td>35</td>
</tr>
<tr>
<td>3.2</td>
<td>fct.ml</td>
<td>35</td>
</tr>
<tr>
<td>3.3</td>
<td>apply.ml</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>f.ml</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>apply.ml</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>Surface type system</td>
<td>46</td>
</tr>
<tr>
<td>4.2</td>
<td>Well-kinding of syntactic tycons</td>
<td>48</td>
</tr>
<tr>
<td>4.3</td>
<td>Semantic type system</td>
<td>49</td>
</tr>
<tr>
<td>4.4</td>
<td>Well-kinding of semantic tycons</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Semantic tycon evaluation</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Surface module language</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>Static entities and entity expressions</td>
<td>59</td>
</tr>
<tr>
<td>6.2</td>
<td>Entity expression semantics</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>Entity declaration semantics</td>
<td>64</td>
</tr>
<tr>
<td>6.4</td>
<td>Functor entity evaluation</td>
<td>65</td>
</tr>
<tr>
<td>6.5</td>
<td>Life-cycle of a tycon</td>
<td>67</td>
</tr>
<tr>
<td>6.6</td>
<td>Relativized type system</td>
<td>67</td>
</tr>
<tr>
<td>6.7</td>
<td>Well-kinding of relativized monotypes, tycons, and types</td>
<td>68</td>
</tr>
<tr>
<td>6.8</td>
<td>Monotype and tycon equivalence</td>
<td>68</td>
</tr>
<tr>
<td>6.9</td>
<td>Type expression equivalence</td>
<td>68</td>
</tr>
<tr>
<td>6.10</td>
<td>Relativization</td>
<td>72</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.1</td>
<td>Map of elaboration processes</td>
<td>75</td>
</tr>
<tr>
<td>7.2</td>
<td>Semantic representations</td>
<td>76</td>
</tr>
<tr>
<td>7.3</td>
<td>Schematic of static environment</td>
<td>78</td>
</tr>
<tr>
<td>7.4</td>
<td>Monotype and tycon elaboration</td>
<td>83</td>
</tr>
<tr>
<td>7.5</td>
<td>Signature elaboration</td>
<td>85</td>
</tr>
<tr>
<td>7.6</td>
<td>Signature spec elaboration</td>
<td>85</td>
</tr>
<tr>
<td>7.7</td>
<td>Signature instantiation</td>
<td>87</td>
</tr>
<tr>
<td>7.8</td>
<td>Top level declarations</td>
<td>88</td>
</tr>
<tr>
<td>7.9</td>
<td>Module declaration elaboration</td>
<td>90</td>
</tr>
<tr>
<td>7.10</td>
<td>Structure expression elaboration</td>
<td>91</td>
</tr>
<tr>
<td>7.11</td>
<td>Signature extraction</td>
<td>94</td>
</tr>
<tr>
<td>7.12</td>
<td>Signature matching elaboration</td>
<td>95</td>
</tr>
<tr>
<td>7.13</td>
<td>Structure signature matching</td>
<td>96</td>
</tr>
<tr>
<td>7.14</td>
<td>Functor signature matching</td>
<td>99</td>
</tr>
<tr>
<td>8.1</td>
<td>Annotated module language</td>
<td>101</td>
</tr>
<tr>
<td>8.2</td>
<td>System $F_\omega$</td>
<td>102</td>
</tr>
<tr>
<td>8.3</td>
<td>$F_\omega$ well-kinding</td>
<td>103</td>
</tr>
<tr>
<td>8.4</td>
<td>$F_\omega$ Type System</td>
<td>104</td>
</tr>
<tr>
<td>8.5</td>
<td>Translation for declarations</td>
<td>106</td>
</tr>
<tr>
<td>8.6</td>
<td>Translation of structure expressions to System $F_\omega$</td>
<td>107</td>
</tr>
<tr>
<td>8.7</td>
<td>$F_\omega$ kind synthesis</td>
<td>108</td>
</tr>
<tr>
<td>8.8</td>
<td>$F_\omega$ tycon synthesis</td>
<td>109</td>
</tr>
<tr>
<td>8.9</td>
<td>$F_\omega$ synthesis of types from normal form semantic tycons, relativized monotypes, and type expressions</td>
<td>110</td>
</tr>
<tr>
<td>8.10</td>
<td>$F_\omega$ type synthesis</td>
<td>111</td>
</tr>
<tr>
<td>8.11</td>
<td>Coercion of dynamic components</td>
<td>112</td>
</tr>
<tr>
<td>A.1</td>
<td>Normalized module calculus NRC</td>
<td>138</td>
</tr>
<tr>
<td>A.2</td>
<td>$F_\omega$-based target calculus TGC</td>
<td>139</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

Acknowledgments. I owe a great debt of gratitude to my advisor, Dave MacQueen, who has guided me and taught me much of what I know about programming language semantics and module system semantics. His detailed feedback was very insightful and critical to the writing of this dissertation. I am also very thankful to John Reppy and Matthias Blume for serving on my committee and their helpful feedback. My special thanks goes out to my early collaborator Robby Findler who pointed me to many opportunities and taught me the “street smarts” of the academic world.

I cannot thank enough the rest of the Department of Computer Science and the Division of Physical Sciences for their tireless and superhuman effort in supporting me in my study. In particular, I would like to especially thank Nita Yack and Anne Rogers of the Department of Computer Science for working miracles. I thank Rick Hefley and Jennifer Woods from the Dean’s Office of the Physical Science Division. My special thanks also goes out to Brooke Noonan of the Graduate Affairs Office. My thanks goes out to Sharon Salveter for her mentoring and guidance for the teaching aspect of my graduate career. I thank Margaret Jaffey for helping me out with the ins and outs of the formal requirements of graduate work. I also thank Donna Brooms for helping me in the numerous practical matters at the University.

My thanks goes out to my officemates Chunyan Song, Mike Rainey, Matthew Jacobs, Adam Shaw, Aaron Turon, Ioan Raicu, and Jon Riehl. Thank you for your good humor and insightful conversations. I also thank the other PLs students Lars Bergstrom and Casey Klein. The TTI-Chicago people were also extremely supportive and in many ways, their interesting ideas, vibrant energy, and feedback helped shape my research and enriched my graduate career. Special thanks are due to Umut Acar, Amal Ahmed, Wonseok Chae, Derek Dreyer, and Xinyu Feng.

I also have to thank the Mobile Software Labs researchers at DOCOMO Labs USA for
inviting me to work on a research internship as part of their FAST Project for Automated Mobile Software specification, verification, and testing. I thank Ajay Chander, Dinakar Dhjurati, and Dachuan Yu.

As for my friends from church, Mary Lin and her family, Paul and Laura Chang, Li Chen, Ningzhong and Ying, Jason and Sue Montgomery, Isaac and Ariel Chen, Howard and Barb Perino, and Frankie Wong, thank you all for inviting me to your home week after week. Your faithfulness encouraged me in most difficult times.

I am very grateful to my aunt who has raised my brother and me. She is always a source of encouragement and inspiration. Thanks to my uncles who are role models. Last but not least I would like to thank my wonderful wife, who was so patient and supportive of my work.
The ML module system is a powerful and expressive language for modular programming and enforcing data abstraction. Several dialects of ML have extended the module system with support for higher-order modules, which improves support for modular programming and elevates the module system to a full functional language. With the exception of Standard ML of New Jersey and MLton, higher-order modules in all these dialects do not obey natural $\beta$-reduction semantics for higher-order functor application (true higher-order semantics). The design space and semantics of a true higher-order module system have not been thoroughly explored. Most of the existing formal accounts of module system semantics neglect true higher-order semantics by separating higher-order functors from type generativity, which limits the flexibility of higher-order functors. The accounts which consider higher-order module semantics neglect to account for interactions between higher-order modules and key core language features such as generative datatype declarations. True higher-order semantics also paradoxically complicates true separate compilation.

In this dissertation, I contribute (1) a novel formal account of a module system with true higher-order semantics, (2) a static entity calculus that cleanly isolates and expresses the higher-order semantics, (3) an exploration of the design space of higher-order module semantics including true separate compilation and the signature calculus, and (4) a translation semantics for compiling the module system to a variant of System $F_\omega$. 
CHAPTER 1
INTRODUCTION

Program modularity is not a radical idea. From early on, programmers have established from experience that breaking programs apart into smaller more manageable pieces is a good idea. Steadily, this idea found its way into the design of programming languages. The ML module system is a powerful language for constructing and abstracting over modules in statically-typed languages. Decomposing a program into modules should not compromise the type-safety guarantees afforded by modern statically-typed languages. The module system itself must have a type system that ensures type-safe composition.

A module system typically partitions code into interfaces (also called signatures) and implementations. Signatures describe the shape of the implementation by listing values and types exported from modules. Implementations rely on signatures to help typecheck occurrences of imported names. A central question in module system design is what should go into signatures. On the one hand, signatures must provide information to typecheck external module dependencies. On the other, signatures should not contain too much information so as to compel the programmer to recapitulate the implementation of the dependency in the corresponding signature.

The main subject of this dissertation is the type system and semantics of an intuitive variant of the ML module system, the true higher-order module system (THO). As I will show, this module system is both type-safe and intuitive in that its semantics is based on $\beta$-reduction. The implementation of the higher-order module system has evolved considerably since the seminal MacQueen-Tofte paper [45] which was the original description of true higher-order semantics. The goal of the rest of this dissertation is to formalize, simplify, extend, and improve the static semantics of true higher-order module systems. This dissertation studies what this true higher-order semantics means precisely and formally by giving a modern formal account. These results open the way to exploration of the design space of
module type T = sig type t end
module Id = functor(X:T) -> X
module type FS = functor (X:T) -> T
module Apply = functor(X:sig module F : FS module M:T end) -> X.F(X.M)
module M0 = struct type t = unit end
module M1 = Apply(struct module F=Id module M=M0 end)
let x : M1.t = ();;

Figure 1.1: Apply functor example in OCaml

functor Apply(X: sig functor F : (X: T) : T structure M : T end) : T

Figure 1.2: Signature assigned to Apply functor under MacQueen-Tofte semantics

*full transparency* (i.e., exactly what can and should propagate through higher-order functor applications) and separate compilation.

### 1.1 Higher-Order Module System

Higher-order modules extend the ML module system with higher-order functors, a natural extension. There are many different applications and three general approaches for higher-order modules [73, 74, 45, 2, 38, 39, 41, 63, 13] which differ mainly on how they handle type sharing in the apply functor (fig. 1.1).

In the apply functor example, the higher-order functor Apply applies functor F to a module M. Tofte [73] introduced the first cut at the semantics for higher-order modules with *non-propagating* functors. Under Tofte’s semantics, the apply functor does not propagate the type M0.t through the functor application X.F(X.M), thus M1.t ≠ M0.t = unit. It assigns the signature in fig. 1.2 to the apply functor. Notice that the signature for the body does not say anything more about *type* t.

MacQueen-Tofte [45] argues that the type sharing M1.t = M0.t = unit should hold. The strong sums model of modules also predicts this behavior of full type propagation [44]. The semantics in MacQueen-Tofte, *true higher-order functors or fully transparent generative*
functor Apply(X : sig functor F : FS structure M : T end) :
     sig type t = X.F(X.M).t end

Figure 1.3: Signature assigned to Apply functor under applicative functor semantics

functors, re-elaborates the functor body of apply given the contents of X.F and X.M at the point of the functor application on line 14. Although this re-elaboration has the desired effect of propagating types, MacQueen-Tofte assigns exactly the same functor signature as the Tofte semantics. Leroy [38] offered an alternative approach, applicative functors, that enriched the notion of type paths in functor signatures with functor applications such as F(M).t. Applicative functor semantics assigns the functor signature in fig. 1.3.

However, applicative functors only solve the type propagation problem under certain circumstances and loses the generative semantics of functors, i.e., functor applications do not generate fresh tycons to enforce abstraction. To address this shortcoming, Moscow ML and Dreyer's module system [16] combined applicative and non-propagating generative higher-order functors in the same language. Fully transparent generative functors both solve the type propagation problem under all circumstances and do not compromise on generative functor semantics. In the last decade, researchers have gained significant experience in engineering non-fully transparent generative functors and transparent applicative functors in compilers such as Moscow ML and OCaml. Although they differ internally, both SML/NJ and MLton compilers support some variant of true higher-order functors semantics. I take OCaml and SML/NJ as representatives of the former and latter groups respectively.

Under both OCaml and SML/NJ compilers, the apply functor example typechecks. Furthermore, although applicative functors cannot directly handle applications to nonpaths, lambda lifting the offending nonpath generally solves that issue. However, if I modify the apply functor slightly by applying a formal functor F to struct type t = int end as in fig. 1.4, then applicative functors fail to propagate enough type information. When the typechecker gets to line 3 in fig. 1.4, it does not have enough information about F to give a stronger signature
module ApplyToInt = functor (F : functor (X:T) -> T) -> F(struct type t = int end)
module R = ApplyToInt(Id)
let x : R.t = 5;;

OCaml functor signature for ApplyToInt
module ApplyToInt : functor (F : functor (X : T) -> T) -> sig type t end

OCaml type error:
This expression has type unit but is here used with type R.t = ApplyToInt(Id).t

SML/NJ functor signature for ApplyToInt
functor ApplyToInt(functor F : (X: T) : T end) : T
structure R = ApplyToInt(Id)
val x : R.t = 5
SML/NJ types x as R.t

Figure 1.4: OCaml and SML/NJ versions of ApplyToInt functor

for ApplyToInt’s functor body. Neither is there a path to the argument struct type t = int end used to construct an applicative functor path. Because the typechecker must give the functor body a signature immediately in order to give HO functor ApplyToInt a complete signature, it can only give the weakest signature, sig type t end. A-normalization gets the program to typecheck but unnecessarily clutters up the code. Thus, in this sense, applicative functors cannot be said to be fully transparent. The SML/NJ compiler has no such restrictions.

The example in figs. 1.4 illustrates the fundamental problem with applicative functors. A higher-order ApplyToInt functor fails to properly propagate types under Leroy’s applicative functor semantics. Consequently, the last line fails to typecheck. They work well as long as the relationship between functor parameter and body is simple, i.e., can be captured in the extended notion of a path with functor application or by A-normalizing the argument. However, not all possible functors fit into this mold. Having to A-normalize arguments of functor applications is an unnecessary shortcoming. Making an argument structure a local definition for the functor application does not help because then after the functor application, the argument structure name would be out of scope. True higher-order functors
propagate types across all functor applications with no change to the source. The solution that MacQueen and Tofte [45] advocate is a re-elaboration of the functor body given the actual argument module.

In instances such as the SymbolTable functor (fig. 1.6), applicative functors admit too much sharing as noted by Dreyer [13]. Applicative functors would permit the symbols from one SymbolTable ST1 to be used to index another ST2 despite the sealing of the functor body to signature SYMBOL_TABLE. Both OCaml and SML/NJ elaborators will make the symbol type abstract, but in OCaml it is only abstract with respect to external clients and not other instances of SymbolTable. Because SymbolTable is applied to the same empty argument for both ST1 and ST2, the two instances also share the same symbol type according to applicative functor semantics. This behavior breaks an important abstraction. Generative functors are more appropriate for enforcing the exact kind of abstraction desired.

In contrast, applicative functor semantics are appropriate for some purposes such as the Set functor in fig. 1.7 where the type sharing of Item.item is desirable and it is acceptable to use items in Sets of the same type interchangeably. However, it is debatable whether the Set functor is a common case. The set of programs T for which true higher-order functors propagate types is exactly those one would want to propagate types. T partially overlaps with the set A the programs such that applicative functor semantics propagates types. In contrast, one should not propagate types in the set A

T. Dreyer [13] noted correctly that applicative functors and non-fully transparent generative
functors are incomparable. Neither applicative nor true higher-order functors can subsume the other. However, there remains the pragmatic question whether the programs that propagate types under applicative functors but not under true higher-order functors ought to have propagated the types in those cases.

The original criticisms of the MacQueen-Tofte semantics are that it lacks support for true separate compilation and that the stamp-based operational semantics makes it difficult

Figure 1.6: SymbolTable functor example from Dreyer [13]

<table>
<thead>
<tr>
<th>signature SYMBOL_TABLE =</th>
</tr>
</thead>
<tbody>
<tr>
<td>sig</td>
</tr>
<tr>
<td>type symbol</td>
</tr>
<tr>
<td>val string2symbol : string -&gt; symbol</td>
</tr>
<tr>
<td>val symbol2string : symbol -&gt; string</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>functor SymbolTable () =</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>type symbol = int</td>
</tr>
<tr>
<td>val table : HashTable.t = HashTable.create (initial size, NONE)</td>
</tr>
<tr>
<td>(* allocate internal hash table *)</td>
</tr>
<tr>
<td>fun string2symbol x =</td>
</tr>
<tr>
<td>(* lookup (or insert) x *)</td>
</tr>
<tr>
<td>fun symbol2string n =</td>
</tr>
<tr>
<td>(case HashTable.lookup (table, n) of</td>
</tr>
<tr>
<td>SOME x =&gt; x</td>
</tr>
<tr>
<td>NONE =&gt; raise (Fail &quot;bad symbol&quot;))</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>structure ST1 = SymbolTable()</td>
</tr>
<tr>
<td>structure ST2 = SymbolTable()</td>
</tr>
</tbody>
</table>

Figure 1.7: Set functor example from Dreyer [13]
to extend the module system and to reason about it. Many recent treatments of ML module
systems abandon true higher-order functors completely due to these issues. The claim is
that the type-theoretic presentations of the module system with applicative functors address
these problems. This dissertation will consider the question whether an operational semantics
account must necessarily be more complicated and if so, why. In contrast to recent work, this
dissertation will take true higher-order module behavior as the starting point for developing
a formal semantics while addressing these criticisms and concerns. My formalism follows
behavior of the SML/NJ compiler which enriches the internal representation of functors and
functor signatures to express the static actions of the functor, thus avoiding a re-elaboration
of the functor body. This approach will also yield some practical benefits. The SML/NJ and
MLton implementations have not kept up with the pace of the progress in module system
design at least partially due to the fact that most of the research has been a radical departure
from true higher-order module semantics. Reframing the state-of-the-art in terms of true
higher-order module semantics will bring recent developments closer a practical extension in
these production-quality compilers.

1.2 Full Transparency and True Separate Compilation

Since MacQueen and Tofte introduced true higher-order modules, many researchers [37, 63,
18] have studied how to downgrade higher-order functors to regain true separate compilation,
_i.e._, Modula-2-style separate compilation. To workaround this limitation, SML/NJ [1, 28]
uses a powerful compilation manager CM [3], which uses a pickled static environments that
include complete static descriptions going beyond syntactic signatures. However, using a
pickled static environment does not solve the true separate compilation problem.

The separate compilation problem can be reframed as a completeness problem for the
signature language, _i.e._, can the source-level signature language adequately describe all possi-
bile modules including functors. Currently, the SML/NJ compiler elaborates module syntax
into internal semantic objects. These semantic objects are expressive enough to encode the functor body relationships that eluded the source signature language. The source and these semantic objects are then compiled to a predicative System F\(_\omega\)-like calculus [68]. This suggests that an F\(_\omega\)-like calculus should be expressive enough to characterize all the static semantic actions of functors. In chapter 6, I describe the entity calculus, a small language that precisely complements the signature calculus and fully describes module types. The semantics is a foundation for future work on the question of separate compilation in higher-order module systems.

Intuitively, true higher-order modules cannot be fully expressed in the syntactic signature language because it is limited to definitional specs and type sharing. For example, there is no way to express a functor signature for the apply functor that accounts for all sharing due to full transparency. Therefore true higher-order modules cannot in general be separately compiled. In particular, HO functor applications cannot always be separately compiled from the functor definition. In the past, various researchers have approached this problem by incorporating applicative functors into the language to varying degrees [38, 2, 63, 16] sometimes limiting the generative functors in the process. In this dissertation, I will argue that applicative functors cannot replace true higher-order functors in the general case. Moreover, if fully transparent generativity is the goal, then applicative functors only serve to support true separate compilation in a limited number of cases.

1.3 Organization

Chapter 2 gives an overview of the evolution of module systems and a discussion of related work including some of the latest variants of the ML module system. Chapter 3 outlines and explores the design space of ML module systems. The focus is on the variants of higher-order functor semantics including applicative functors, type generativity, first- and second-class modules, separate compilation, and signature calculus design. It discusses the implications
of true higher-order semantics with respect to the other major features of the module system and compares the design with other approaches to higher-order functors and considers the implication of true higher-order semantics in light of the goal of separate compilation.

Chapter 4 introduces the type system for the surface language. The type system is a first-order calculus supporting type constructors. The chapter covers the kind system and static semantics of the type systems. Chapter 5 defines the core and module surface (i.e., syntactic) languages. The discussion covers important classifications of type constructors in the context of the module language.

Chapter 6 discusses the entity calculus, a small functional language that precisely defines the functor actions needed to fully express true higher-order semantics. The chapter gives a evaluation semantics for the entity calculus and establishes some key properties along with the semantics for the relativized type languages. The next part of the dissertation formally defines the semantics of the module system in two interweaving modes. Chapter 7 introduces the elaboration semantics that both constructs entity calculus expressions and typechecks programs using the result of the evaluation of those expressions.

Chapter 8 defines a system that translates module syntax to an enriched variant of System $\text{F}_\omega$ and establishes the soundness of the elaboration semantics based on the soundness of System $\text{F}_\omega$’s type system.

Chapter 9 outlines some directions for future work and concludes. Appendices A and B discuss semantics and implementation of higher-order modules in SML/NJ and contrasts it with that of the present work. Appendix C provides the proofs of the formal properties in above chapters. Finally, Appendix D contains an index of the technical notations used in the body of the dissertation.
CHAPTER 2
BACKGROUND

Modularity in programming dates back at least to Parnas and his information hiding criteria[55]. There are two main lines in the development of module systems. First, the Modula [77] and Cedar/Mesa [24] line emphasized ad hoc and extralinguistic schemes for modeling linking of program modules, i.e., such languages used a separate tool, an object file format sensitive link-loader, to compose separately compiled modules. Second, the ML line of modularity [42, 44, 43] described modularity in terms of a small functional programming language where application is the main linkage mechanism. Common to both of these approaches is the idea that the separation of interface and implementation leads to better program structure.

The original motivation for program modularity was highly pragmatic. Computers with very small main memory could not compile a large program efficiently because program source could not fit into memory all at once. Early languages partitioned programs into separate compilation units so that each individual compilation unit can mostly fit into memory and therefore be amenable to efficient compilation. As machine architecture progresses, program modularity is of increasing importance. Because most compiler optimization techniques rely on delicate data and control flow information and thus are sequential, modern compilers are rarely parallelized beyond the compilation of program modules in parallel. Compilers such as distcc can only take advantage of multiple cores during compilation if programs are broken down into finely separated compilation units. Thus, modularity is necessary for efficient compilation even on modern architectures.

Modules in ML may include type components. This provision affords the programmer considerable power and fine-grain control. The ML module system serves three main functions:

1. hierarchically organizes source (namespace management)
structure A =
struct
  structure B =
    struct
      structure C = struct type t = int end
    end
  end
end

type u = unit end

2. provides a mechanism for generic programming through parameterization
functor F(X: sig type t end) =
struct
  type u = X.t
end

3. enforces abstraction boundaries
structure M = struct type secret = int end :> sig type secret end

The most basic form of modularity is namespace management. Simple namespace management only requires a means to declare namespaces at the top-level of program structure (namespace N), functions that reside inside those namespaces, and a notation for accessing members of one namespace from another (N.f). Each namespace is a module in such a system. One can neither manipulate nor reference a namespace outside of the projection notation. A programmer can call functions by projecting out the desired components from these rudimentary modules. Namespace management, however, is only a convenience for the programmer. It provides no additional expressiveness or power in the language. In particular, namespace management by itself does not support separate compilation, which is crucial for independent development of components of a large software project and efficient compilation. In fact, without separate compilation, some programs would have been impossible to compile with the limited resources on early machines. Even on modern machines, large software compilation can be quite taxing. Many languages did not gain even a rudimentary module system until the late 1980s and early 1990s. For example, the venerable FORTRAN did not include a rudimentary module system until FORTRAN 90.

A record of functions and variables can be used as a slightly richer form of modularity. In functional languages, regular records may also contain functions as fields. Records aggregate
labeled fields that can be projected. Unlike namespaces, records are typically first-class, so
functions on records enable a degree of generic programming especially in conjunction with
row polymorphism [59]. Unfortunately, records normally do not support data abstraction.
Pierce’s model of object-oriented languages uses records as its core construct [57]. Lampson
and Burstall’s Pebble language featured heterogeneous records that contained both values
and types [35]. Cardelli’s Quest and Fun languages couple System F-like languages with
heterogeneous records [7].

2.1 Module Systems in Other Languages

Modula-2 pioneered many of the major ideas behind module systems. It supported hierarchi-
cal composition (i.e., nesting), type components, and an exception handling mechanism [77].
The exception handling mechanism consisted of exception declarations as components of
modules and a module-level exception handler. Compositions of modules were limited to
explicit imports and exports of definite modules (as in a specific concrete module with all
its components filled out) and their components.

Ada is an example of an imperative and object-oriented language that supports mod-
ularity in the form of basic and generic packages [6]. Packages are specified in two parts,
specifications (akin to interfaces or signatures) and bodies which define the implementation
of the package. Generic packages map basic packages to basic packages, thus provide a form
of parameterization. Basic packages can be nested. Ada supports type components in mod-
ules (called packages) and “generic” modules parameterized on a single restricted type and
some associated operator definitions.

Object-oriented languages use the class or object system to provide a degree of modularity
for functions and data. Most object-oriented languages with the exception of Scala [52] do
not support type components, thus limiting their expressiveness. Scala’s object system
supports type components as members of a class. Beyond simple namespace management,
OO languages support information hiding via private data members. Unlike in ML module systems, information hiding is achieved by annotating member fields and functions with a label private or protected instead of hiding type definitions. Unfortunately, this is a rather coarse-grain encoding even with multiple levels of privacy, visibility to an external client is all-or-nothing. Scala’s structured types offer another approach to modularity. There has also been a considerable amount of interest in Bracha’s mixin approach to modularity [5, 4, 19] and traits [66, 21].

Type classes [76] can be modeled using a very stylized form of modules coupled with a dictionary-passing type inference [17]. Where type classes apply, this inference scheme makes code significantly more succinct. However, as the set of instances of classes at any one point in the program is global, type classes sometimes interfere with modular programming.

### 2.2 The ML Module System

The ML module system consists of structures, signatures, functors (parameterized modules), and functor signatures. Structures are sequences of type, value, structure, and functor bindings. In Standard ML, structures can be named or anonymous. Signatures are sets of specifications for types, the types of values, structure component signatures, and functor component signatures. Similar to structures, signatures can be either named and anonymous. Anonymous signatures are either written inline (i.e., `sig ... end`) or extracted (inferred) from a structure. There is a many-to-many relationship between structures and signatures. Ascribing a signature to a structure verifies that that structure satisfies the signature. Alternatively, one can use a signature `SIG` to seal a structure `M` which hides or reveals type definitions selectively as required by `SIG`. Type specifications in signatures can be transparent definitions (`type t = unit`), open (`type u`), or translucent specs defined relative to open specs (`type v = u * int`), usually considered a kind of transparent specification.

The module system of ML can express much of System F. Indeed, the SML/NJ com-
piler compiles source into a System F-like core language (see chapter 8). Although the Definition[47] does not require nor define support for higher-order modules, they can be found in many implementations of Standard ML including SML/NJ, Moscow ML, and MLton. Higher-order module systems are not found outside of the ML family of module systems. Functors can be typechecked independently from their uses (i.e., applications) unlike related programming constructs such as C++ templates. Separate typechecking is necessary for true independent development of large software systems. It also makes it possible to gain confidence in the type safety of the functor before even beginning to write functor arguments and applications of functors, thus identifying type errors earlier in the development process when they can be resolved more easily. The ML module system also supports type abstraction via opaque ascription. With this feature, programmers can define abstract data types and data structures with their corresponding suite of operations.

Another important property of ML module systems is the phase distinction [30]. If a language respects phase distinction, then the static and dynamic parts of the program can be separated such that that former does not depend on the latter. Because the ML module system can be compiled into System F$_\omega$, it naturally respects the phase distinction. At first glance, it may appear that types and values are entangled in type definitions such as type $t = M.u$ where M is a module that contains value as well as type components. Because type components always only refer to other type components in modules, one can split any module into one that only contains type components and one with only value components. The proviso is that modules cannot be first-class. Because ML stratifies the module and core languages and treats modules as second-class, this proviso holds.

Comparisons of ML Module System and Other Languages

Garcia et al.[23] evaluated how well 8 major programming languages support the implementing type-safe polymorphic containers by generic programming techniques. Their chief
criticisms of the ML module system are three-fold. They argue that the signature language should permit semantic compositions of signatures beyond the syntactic include inheritance mechanism similar to Ramsey’s signature language design [58]. The authors also remark that the module system would be impractical for programming-in-the-small because the syntactic overhead is excessive even when compared to other languages that require explicit instantiation. The lack of “functions with constrained genericity aside from functions nested within functors” was also cited as a disadvantage. Garcia et al. suggests that some bounded polymorphism would be desirable.

2.3 Separate Compilation

A signature language supporting separate compilation needs a mechanism to ensure coherence of the abstract types in imported modules. ML presently solves the coherence problem through a combination of type sharing constraints, definitional specs, and where type clauses. In simpler module systems, the problem of coherence is not as pronounced because compilation units only import external units using definite references [71]. Sometimes the module system needs to create a dummy structure from a signature in a process called instantiation. Instantiation constructs the free instantiation of a functor formal parameter that imposes the required amount of type sharing but no more. When instantiating a signature, type sharing constraints must be resolved. This process is a complicated with a considerable amount of folklore. The instantiation phase in SML/NJ imposes a semantics stricter than that of the Definition. Although Harper and Pierce [31] claim that type sharing constraints can be superseded by definitional type specs in all cases, my study has identified examples (fig. 2.1) where this is not the case absent a mechanism for signature self-reference, e.g.,

```
structure M0 : S0 where type u = self.M1.s.
```

Since type w depends on both M0.v and d, the definitional spec cannot be moved. Moreover, the enclosing structure M1 must come after M0 to ensure that M0 is in scope. Datatype s depends on type w, hence it must come after the
Both Russo [63] and Swasey et al. [71] have suggested that the separate compilation problem can be solved by identifying an alternate form of compilation unit, one that is not a module. OCaml, in fact, already implements such a regime. However, in the presence of true higher-order functors, these approaches merely punt on the real problem by compelling the programmer to put otherwise independent modules together in a single compilation unit and then abstracting over that unit. In the end, the problem of separate compilation in a true higher-order module system is a design question. Enriching the module system’s signature language to such a point that it can express true higher-order semantics results in a rich language of higher-order type constructors and static effects, \textit{i.e.}, \textit{type generation}. At this point, the question becomes whether it is reasonable and practical to have to programmer recapitulate in the signatures so much of the higher-order behavior already in the modules.

2.4 Formal Accounts of ML-based Module Systems

Module systems have generally followed either in the extralinguistic style of Modula and Mesa or the functional style of ML. Dreyer and Rossberg’s recent module system MixML [18] leans towards the extralinguistic style with the subtle mixin merge linking construct that serves many roles including encoding functor application. Cardelli [8] brought some
amount of formalization to this *ad hoc* approach to modularity, but module systems in this style still vary considerably in terms of semantics. Moreover, the extralinguistic semantics of linking are typically fairly involved and delicate.

The ML module system has inspired a bevy of formalisms describing its semantics and common extensions. These formalisms run the gamut from Leroy’s presentation based on syntactic mechanisms alone, to Harper-Lillibridge [29] and Dreyer, Crary, and Harper’s [16] type-theoretic approach, to the elaboration semantics approach as represented by the Definition [47]. As I have discussed at length in sec. 1.1, full transparency is a desirable property for module system designs. Designs range from no support for transparency (type propagation) such as non-propagating functors to complete support for MacQueen-Tofte full transparency. In between are the various gradations, *i.e.*, different combinations of applicative and generative functors. Another important issue in module system design is the formalism used for expressing the semantics. In the early days, module system semantics were quite *ad hoc*, each relying on their own peculiar notation and collection of semantic objects [77, 24, 73]. Since then, there has been a move toward more standardized logical frameworks that are more amenable to encoding in a theorem proving system such as Isabelle/HOL or Coq [29, 16]. Curiously, the most recent accounts have been moving back towards a semantic objects-based approach [14, 18]. Fig. 2.2 shows very roughly where the major families of module system designs fall in relationship to each other in terms of support for full transparency and adherence to some formal logic framework. In the figure, it appears that the left-hand side has been well-explored. I will study the right-hand side, and perhaps the upper-right-hand quadrant which represents a fully transparent semantics in something close to a mechanized metatheory in, say, Coq. This study will ideally result in a module system design that reconciles a current understanding of fully transparent functors with a simpler and more accessible semantics, a clear evolutionary step from the state-of-the-art.

Besides the main module system features described earlier in this proposal, there are a
A semantics is more standard in the sense that it uses more formal logic frameworks. It may potentially be easier to mechanize the metatheory of such semantics.

Figure 2.2: The spectrum of major module system families

number of other useful properties developed in the different module languages. The *phase separation* [30] plays a fundamental role in ensuring that module systems can be typechecked fully at compile-time. This property is desirable, being consistent with our goal of static type safety. It says that a language, in this case a module, can be split into static and dynamic parts such that the static part does not depend on the dynamic. Some accounts of module systems respect phase distinction at the surface language level [38, 63]. Others respect the phase distinction in an internal language but not the surface language [45].

The other key issue in a module system design is the existence of a *principal signature* for modules. The term principal signature has been overloaded in meaning. Because of the many-to-many relationship between signatures and modules and the signature subtyping relationship, many signatures may be safely ascribed to a single structure. I will call the most precise signature for a structure (i.e., one that constrains all components of the structure with exact, most constraining types) the *full signature*. A related concept is the *free instantiation*
of a functor formal parameter. The free instantiation is an instance of the functor formal parameter signature \( S \) that admits exactly enough type sharing to satisfy \( S \) and no more. In particular, it avoids any extraneous type sharing that would cause potentially distinct types to be shared. What Dreyer [13] calls the principal signature is the full signature in this terminology. Tofte defines principal signature of a signature expression \( \text{sigexp} \) as one whose formal tycons can be instantiated to obtain all instantiations of \( \text{sigexp} \) [73, 47]. I will adopt Tofte’s terminology.

### 2.4.1 Syntactic Approach

CMU School

Beginning with Harper-Lillibridge’s translucent sums module calculus [29], this large, prolific family of module systems pushed the state-of-the-art in terms of the type theoretic approach to module system design. Although Harper-Lillibridge originally explored first-class modules, the bulk of the research in this family was directed towards an applicative higher-order module semantics for type-directed compilers TIL/TILT and adding recursion.

Crary et al. [9] and later Dreyer [11, 14] (DCH) have explored adding support for recursion. Harper-Lillibridge [29] and then Russo [64] studied mechanisms for making modules first-class entities in the core language. With first-class modules, programmers can leverage the familiar module system to take advantage of the System F-like power of the module system for programming-in-the-small. Unfortunately, as Garcia [23] remarks, the syntactic overhead of the ML module system makes this undesirable and impractical. When used in a similar context, Garcia suggests that the type inference used for type classes makes modular programming-in-the-small more succinct. This observation holds only in the specialized case of type classes.

Harper and Stone [32] gave a semantics for ML modeling every core language construct
in terms of modules. This semantics was based on the technique of using singleton signatures (and types) to express type definitions [70]. Dreyer’s semantics uses a signature synthesis algorithm that constructs a signature directly from a structure expression syntax given the static environment as context. In contrast, THO constructs a signature solely using the static environment.

Syntactic Paths and Applicative Functors

One of the first formal accounts of an ML-like module system is Leroy’s manifest types calculus [37] where manifest types are definitional type specifications. The surface language for the manifest types calculus is equivalent to that of the translucent sums calculus described by Harper-Lillibridge. The key observation in the manifest types calculus is that one can typecheck manifest types by comparing the rooted syntactic paths to those types which uniquely determine type identity. Thus, type equivalence is syntactic path equivalence for type constructors. Leroy introduced the notion of applicative functors [38] which held types in the result of a functor application to be equivalent to corresponding types in all other results of applications of that functor to the “same” argument. Several designs [69, 16, 63] have tried to incorporate both applicative and generative functors in a single calculus. Cregut [10] enriched signatures with structure equalities to obtain complete syntactic signatures for separate compilation.

Leroy introduces a relatively simple approach to module system semantics that precludes shadowing of core and module bindings [39, 40]. The semantics supports type generativity and SML90-style definitional sharing by reducing them to solving path equivalence by way of A-normalization (for functor applications) and S-normalization (a consolidation of sharing constraints by reordering). In his module system, Leroy claims that all type sharing can be rewritten in his calculus with generative datatypes and manifest types. However, Leroy’s

---

1. Definitional specs appeared in SML/NJ 0.93 a year before the Harper-Lillibridge paper appeared.
simplified module system does not include value specifications and datatype constructors both of which can constrain the order in which specifications must be written and therefore result in situations where sharing constraints cannot be in general reduced to manifest types.

For full transparency, Leroy proved that there is a type-preserving encoding of a stratified calculus with strong sums without generativity using applicative functors\[38\], claiming that the existence of such an encoding is a strong hint that applicative functors support full transparency. My HO apply functor example in fig. 1.4 casts some doubt on this claim. As Leroy pointed out, under the strong sums model, first-class modules are at odds with phase distinction because of the typechecker would have to do arbitrary reductions\[37\]. In contrast, because the weak sum model of the manifest types calculus does not require any reductions at typecheck time regardless of the presence of first-class modules, it does not violate the phase distinction\[37\]. In the most recent paper in the manifest types series \[40\], Leroy abstracts away most of the core language details from the manifest calculus to obtain a mostly core language independent module system.

Although the syntactic paths approach may very well provide the simplest account of module systems, this is at the cost of some very fundamental shortcomings such as the inability to support shadowing and full transparency. Because the account’s support for type sharing is incomplete, there may also be limits to how the semantics deals with the coherency issue. This dissertation will address these issues which are fundamental to the power of the ML module system.

Other ML Variants

Shao \[67, 69\] defines a signature language based on gathering (and internally factoring out) all flexible components (i.e., open specs unconstrained by sharing) in a higher-order type constructor that can be applied to obtain a signature that expresses functor body semantic actions at a later point. The resultant signature language superficially resembles applicative
functors. However, type constructor applications in the signature language must be on paths. Consequently, it does not support full transparency in the general case.

Govereau [26] developed an alternative semantics based on the type-theoretic approach. He claims that there is no declaration that produces a generative type that is not abstract, but datatype declarations could potentially be one of these cases. One observation Govereau makes is that the static environment must be ordered because types are dependent. Later bindings in the environment may refer to earlier bindings. The combination of applicative and generative types leads to a soundness problem similar to that found in Russo’s Moscow ML system.

The main contribution in Govereau’s semantics appears to be a syntactic way to represent generativity that obviates the need for a Dreyer’s effect system for managing the interaction between applicative and generative functors. The language uses distinct declarations for generative and non-generative abstract types. The type system declaratively partitions the static environment into two, corresponding to the lexical scope inside and outside a functor.

The paper claims that if a type is defined “under functor” but does not depend on a generative type defined under a functor, then it can be coerced to non-generative. The claimed win is that although the syntactic purity judgment is not as fine-grain as DCH’s effect system, it is simpler to work with. One disclosed limitation to both Govereau and Russo’s systems is that a non-generative functor cannot have any generative types that are used in the body of the functor.

Alice ML provides a number of the features mentioned in this chapter especially in the area of signature language enrichments. The language supports nesting signatures inside of signatures.

\[
\text{signature } S = \text{sig signature } M = \text{sig type } t \text{ end end}
\]

Furthermore, the nested signatures can be left open to support signature-polymorphic functors.

\[
\text{functor } F (X:\text{sig signature } S \text{ structure } Y : S \text{ end}) = ...
\]
The abstract signatures do not, however, appear to be complemented with any bounded polymorphism features.

MLKit’s semantics for elaboration calls for what amounts to defunctorization. In Elsman’s view, functors are purely static constructs that can be compiled away, generating inlined copies of relevant dynamic content upon functor application.

Dreyer and Rossberg [18] show how to encode ML signatures, structures, and functors in a mixin module calculus that appeals to something similar to Bracha’s merge[4] as its only linking mechanism. When linking a module A and B, the semantics tries to satisfy the imports of A using the exports of B and vice versa. The mixin merging syntax is \texttt{link x=M0 with M1} in the surface language. It binds the name x to a module M0 and concatenates it with M1 merging components where appropriate. The scope of x is the body of M1. This mixin merging semantics supports recursion and separate compilation. Modules in this language consist of atomic modules that only contain values, types, type constructors, etc., labeled modules ($\{\ell = M\}$), and the merging form \texttt{link ... with ...}.

The peculiarity in this language is that modules and indeed anything that can be encoded in these modules are stateful. For example, signatures in MixML are fundamentally stateful. Linking against a signature S mutates it. Consequently, the typechecker rejects the following program.

```
module U = link x={module S = {type t}}
with {module A=(link x=x.S with {type t = int}),
      module b=(link x=x.S with {type t = bool})};
```

Signatures must be suspended and then new’ed in order to be matched multiple times. This suspension is called a unit in Dreyer and Rossberg’s terminology. Functors are also represented by suspending modules with unsatisfied imports. Although units have full support for hierarchical composition, MixML’s design still retains the problem of stratifying modules and units, a problem inherited from the Flatt-Felleisen units that inspired it.

As it stands, the part of the MixML language that encodes the ML module system is
\( f \) is a fresh higher-order dependency variable

\[ \Gamma, \mathcal{W} \vdash \text{type } t \Rightarrow ((t \mapsto f(\mathcal{W}), \emptyset), \{f\}) \]

Figure 2.3: Biswas’s elaboration rule for abstract type specifications: \( \Gamma \) is the type environment mapping program variables to types. \( \mathcal{W} \) is a list of formal parameters variables the specification may depend on.

but a small fraction of the whole. The question remains what implications the rest of the language has. Part of the language is obviously semantically meaningless such as \textbf{link} \( X = [\text{int}] \) with \[3\], which is a well-formed program.

Moscow ML

Biswas gave a static semantics for a simpler form of higher-order modules [2]. The account relies on semantic objects and a stamp-based semantics similar to the Definition. The type propagation in higher-order functors is captured by a “higher-order” dependency variable that abstracted possible dependencies on the argument. These variables are only present in the internal language produced during elaboration. Consequently, Biswas’s semantics does not support true separate compilation and neither does it enrich the surface syntax for signatures. Biswas’s elaboration rule in fig. 2.3 maps an abstract type name \( t \) to the fresh higher-order abstract dependency variable \( f \) applied to the list of all abstract dependency variables \( \mathcal{W} \) it could possibly depend upon. For example, the functor parameter of the Apply functor, \textbf{functor} \( F(\text{X: sig type t end}) : \text{sig type t end} \), is given the semantic representation

\[ \forall f_0(\{t \mapsto f_0\} \Rightarrow \{t \mapsto f_1(f_0)\}). \]

Biswas’s formal account was extended in a somewhat more type-theoretic style to the full SML language and implemented by Russo in the Moscow ML compiler [63]. One semantic innovation was the use of existential types to represent abstract types in a module system. The idea of using existential types originated in a paper by Mitchell and Plotkin [48]. MacQueen pointed out how scoping unpacks make conventional existential types ill-suited for modular programming [44]. Montagu and Rémy [49] recently tried to address these concerns.
by developing a variation of existential types which replaces unpack with modular open and scope restriction constructs.

Moscow ML also adds support for first-class modules[64] and a form of recursion [65]. Moscow ML further attempts to incorporate both non-propagating generative functors and applicative functors. The main limitation, as pointed out by Dreyer [13], is that Moscow ML’s combination of generative and applicative functors is unsound. In particular, any generative functor can be \( \eta \)-expanded into an applicative functor thereby circumventing the generative functor abstraction.

Russo notes the conflict between true separate compilation and first-order module systems. He works around the problem by defining compilation units to be only the sealed modules in an applicative higher-order module system with first-order generative functors. To address the avoidance problem, he introduces a hybrid name and deBruijn-indexing scheme.

Rossberg, Russo, and Dreyer [62] developed a formal semantics based on a translation to System F\(\omega\) enriched with records and existential types in a recent variation on Russo’s type system. In this account, the type system of the module system was exactly the type system of their variation on System F\(\omega\). All semantic representations of signatures explicitly existentially quantified all primary type constructors. The module system supported type generativity insofar as existentials support it. The module system does not claim to support full transparency, although there is an extension of the system to support applicative functor semantics.

2.4.2 Units and Other Extralinguistic Linking Systems

Flatt-Felleisen [22] and Owens-Flatt [54] develop a module system semantics based on a calculus with stratified distinct hierarchically composable modules and recursively linkable units. Because both accounts appeal to extralinguistic linking semantics, they fall under
the Module/Mesa line of module systems. As pointed out by Dreyer[18], the fundamental limitation in this semantics is that the stratification of units and modules precludes using unit linking and hierarchical composition together. One strength of their module system design is that it is one of the few accounts that includes an operational dynamic semantics, unlike all the other accounts discussed in this section. Some accounts of module systems give only a static semantics and perhaps a typechecking algorithm. Owens and Flatt prove type soundness of their semantics.

Swasey et al. [71] described a calculus SMLSC that is modeled after Cardelli’s linkset approach to separate compilation. SMLSC introduces a compilation unit that sits on top of the module system that can be separately compiled from unimplemented dependencies by means of a handoff unit whose role resembles that of header files in C.

2.4.3 The Definition and MacQueen-Toft

The module system semantics in the Definition of Standard ML [46, 47] evolved throughout the late 1980s and early 1990s. Early on, Harper et al. gave a fairly complete account of the static semantics of the first-order ML module system in terms of an operational stamp-based semantics [27]. Tofte in his thesis [72] proves that signature expressions in the first-order, generative semantics have principal signatures. He also extended this proof to cover non-propagating higher-order generative functors [74]. Then MacQueen and Tofte [45] introduced true higher-order functor semantics.

Apart from type propagation transparency issues, the evolution of the Definition also addressed type sharing issues. The role of type sharing constraints has evolved through the development of the Standard ML semantics and SML/NJ implementation. Type sharing constraints solve two problems in ML, type specification refinement and coherence. Originally, type specifications only declared the name of an expected type. It is quite useful to be able to refine type specifications to restrict them to particular definite types or by
defining them relative to open specs. The *coherence problem* is the challenge of constraining type components of two structures to be equivalent regardless of the actual identity of that type. In SML90 [46], explicit sharing equations among visible abstract type constructors and generative structure sharing were the sole means for constraining type specifications. Under generative structure sharing semantics, each structure had a unique static identity. Thus, two structures shared only when they were identical in the sense that they were defined at the same point in the program and are merely aliases (fig. 2.4). Under SML90 identity-based structure sharing, A and B have to be aliases, so the functor application at M4 fails to typecheck. Under SML97, the sharing constraint merely rewrites to sharing type $A.t = B.t$, thus both functor applications typecheck. Structures that shared in SML90 were equivalent both statically and dynamically. This kind of rich sharing semantics turned out to be quite complicated and was abandoned in favor of a structure sharing that implied type sharing constraints on congruent type specifications within the respective signatures.

SML/NJ 0.93 introduced definitional specifications, giving programmers two ways for constraining types to definite ones. SML97 added where type and definitional type specifications completely replaced the definitional type sharing found in SML90. Definitional specifications and type sharing were finally disentangled. Generative structure sharing was eliminated in favor of the simpler semantics of a structural sharing that amounted to a type sharing equation for each common type component path, no matter how deeply nested. The semantics of type sharing and related mechanisms such as where type are still somewhat problematical and unsettled [50, 58]. Ramsey *et al.* have argued that the scoping of where type definitions should be more symmetric thereby permitting more flexible type specification refinements.

Shao [68] (in a paper unrelated to the one on applicative functor-like module system [69]) extends MacQueen-Tofte fully transparent modules with support for type definitions, type sharing (normalized into type definitions), and hidden module components. This treatment
signature S0 =
  sig
    type t
    val f : t -> t
    val state : t ref
  end
functor F(X: sig
  structure A : S0
  structure B : S0
  sharing A = B
  end) = ...
functor G() =
  struct
    type t = unit
    val f = ...
    val state = ref 0
  end
structure M0 = G()
structure M1 = G()
structure M2 = M0
structure M3 = F(
  struct
    structure A=M0
    structure B=M2
  end)
structure M4 = F(
  struct
    structure A=M0
    structure B=M1
  end)

Figure 2.4: SML90 structure sharing example
of higher-order modules is a more recent form of what is currently in the SML/NJ compiler. Elsman presents a module system compilation technique used in the ML Kit compiler [20]. The semantics follows the style of the Definition. The compilation technique is comparable to Shao’s FLINT compilation scheme.

The SML/NJ compiler implements a version of the module system that departs from the Definition in a number of aspects. Some of these extensions have not been formalized as of yet. In particular, the compiler has a richer semantics for include and the elaborator compiles a functor body to a static lambda calculus which is what is used in place of the functor body re-elaboration at functor applications. The scoping of sharing constraints has also changed. In the current implementation, SML/NJ no longer permits nonlocal forms of sharing of the flavor illustrated in the example in MT (the first structure sharing constraint in fig. 2.5). Both sides of the constraint must be in local scope as is in the second sharing constraint. Instead, structure definition specs express the same kind of sharing. For example, the structure definition spec structure A : sig end = S expresses the first sharing constraint in SIG. The module system also has some significant limitations such as a lack of recursion, the tension between separate compilation and fully transparent generative functors, and weak facilities for signature composition.
2.4.4 Type Generativity

The idea of generative types has a significant history. More recently, Neis et al [51], Dreyer and Blume [15], Dreyer [12], and Rossberg [60] have all developed semantics for studying type generativity in the context of a System F\(_\omega\)-derived language. Although the ML module system utilizes type generativity, it only plays a role within the module system proper. After translation, at which point the modules are eliminated, all the type generation has already occurred, having been carried out by the elaborator. Consequently, the target language for translation, a System F\(_\omega\) language, need not contain any facilities for generating fresh types.

2.4.5 First-class polymorphism inference and type classes

Jones [34] motivated first-class polymorphism (FCP) by appealing to the constructive logic tautologies for existentials, universals, and implication [34]. The constructive logic rule

\[
\langle w, \tau_w \rangle \rightarrow \tau' \leftrightarrow w \Rightarrow \tau_w \rightarrow \tau'
\]

corresponds to a translation of a module expression where types and values are coupled to System F\(_\omega\) where types and values are decoupled (phase separation) in the forward direction. More recent first-class polymorphic calculi such MLF [36] and FPH [75] add some limited type inference. Although inference in general may be undecidable, these limited inferencers still go a long way to make programming in these first-class polymorphic calculi more practical. If some of the ideas for inference for FCP calculi can be transferred over to a module calculus, one might also address the syntactic overhead of ML module systems. Adding FCP to the core introduces a certain amount of redundancy with respect to the FCP afforded by the module system. It would be useful to consider what exactly is redundant and whether that can be minimized.

Another language construct related to module systems that enjoys type inference is the type class. Type classes are a special case of modular programming where a kind of automatic deduction would be useful. Unfortunately, the scope of class instances are global. In Modular Type Classes, Dreyer et al. [17] develop semantics and a translation from type classes to a
stylized use of the ML module system.

2.4.6 FLINT/Typed Cross Compilation

The most closely related work is the formal semantics for a module language called NRC given by Shao [68]. Similar to this dissertation, NRC gives a formal semantics based on MacQueen-Tofte’s higher-order semantics. Unlike the present account, Shao’s semantics combines elaboration and translation into a single interweaving set of judgments, whereas the present account cleanly separates them. The separation of elaboration and translation is a conceptual simplification and is also beneficial for more modular compiler implementation. Also, Shao’s semantics assumes a preprocessing step that eliminates most of the interesting behavior involving coercive signature matching. Although the NRC account alludes to the fact that the functor body does not need to be re-elaborated at each application, the NRC semantics does re-elaborate much like the MacQueen-Tofte account. Finally, the NRC semantics does not directly support datatype declarations, though it is claimed that it is a straightforward extension.

The account of NRC makes a few arguable simplifying assumptions. Value components is a trivial extension that only requires a conversion of source-level types to semantic types. Signature extraction is trivial only requiring the conversion of semantic types back to syntactic types. This conversion is guaranteed by the 1-to-1 mapping of the stamp environment. This is similar to our entity environment (see chapter 6), but as the semantics in demonstrate, this is far from trivial. Shao says that value components in NRC are only elaborated once, but any type components may be re-elaborated. We simplify this even further. Only functor applications and type generation have to be re-elaborated. Type definitions can be expanded out early on during elaboration. As with some other accounts, NRC entangles type-checking and translation into $F_\omega$. Although translation clearly depends on type-checking, the converse case is not true. In NRC, the assumption is that type-sharing constraints can all be
converted to type definitions. As I have shown earlier, this is technically incorrect but probably practically adequate. NRC requires A-normalization of the surface syntax SFC so that the only functor applications are of a functor variable (not paths) to a structure variable. In THO, the operator position of applications can be a path and the operand can be any structure expression. The $\alpha$-renaming scheme described in NRC adds new declarations inside of existing structures in order to deal with signature ascription. Signature matching is mostly handled during the SFC to NRC translation by adding explicit enrichment coercions. NRC relies on a stamp environment, a finite map from stamp to tycon path. This has turned out to be unnecessary due to the existence of a function to perform the equivalent operation while relying on only the present semantics’ structure entity itself. This function is the moral equivalent of an inversion of the realization although the realization does not technically admit an inversion.

2.4.7 Summary

Not all module system designs enjoy the principal signature and phase distinction properties. Fig. 2.6 summarizes the key features of the main ML-like module system families. Note that many module systems have various combinations of these features, but none are complete. Ideally, a module system would have true higher-order semantics and all the other features except for applicative functors which would be redundant.
<table>
<thead>
<tr>
<th>System</th>
<th>higher-order</th>
<th>first-class</th>
<th>sep comp</th>
<th>rec</th>
<th>app</th>
<th>gen</th>
<th>phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>HL [29]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Leroy [38]</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Russo [65]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DCH [16]</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RMC [14]</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MT [45]</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MixML [18]</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ideal</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

higher-order = true higher-order  
rec = recursive modules  
app = applicative functors  
gen = non-propagating generative functors  
phase = respects the phase distinction

Figure 2.6: A comparison of major ML-like module systems
CHAPTER 3
THE DESIGN SPACE

The simplest ML module system features only hierarchical composition of structures, type components in structures, transparent signature ascription, and first-order functors. Beyond this set of features lies a design space subject to debate. The module system design space is a rich one consisting of several functional dimensions. Much of the design space revolves around the module system’s abstraction enforcement through opaque sealing and how that interacts with the other features of the module language.

Dreyer frames the design space in terms of phase-separability and purity with respect to static and dynamic effects. In contrast, the goal of the THO semantics to focus on a simple framework for combining higher-order functors and type generativity. Type generativity is the central issue in the THO semantics. Although the semantics does not add anything new to the discussion of first-class and recursive modules, it appears that it neither complicates nor simplifies the issues concerning these two features.

3.1 Applicative Versus Generative Functor Semantics

A chief point of contention is the support of applicative versus generative semantics for functors and the various attempts at supporting both. The central argument of applicative functor semantics is that a functor applied multiple times to the same path should produce results with the same type components. The original motivation was the support of higher-order functors with syntactic signatures. Subsequently, researchers have found few other examples where applicative functors have proven to be useful. One frequently cited example is the set functor (fig. 3.1). The claim is that one would want to apply the set functor multiple times to the same argument, opaquely ascribe each result, and still expect each result’s corresponding types to be equivalent.
functor SetFn(K: sig type t ... end) =
structure
  type set = K.t list
  val empty = []
  fun isEmpty = ...
end

sig
  type set
  val empty : set
  val isEmpty : set -> bool
end

structure IntKey = struct type t = int ... end
structure I0 = SetFn(IntKey)
structure I1 = SetFn(IntKey)
I0.isEmpty(I1.empty)

Figure 3.1: Set functor

module SetFn(X: sig type t end) :
sig
  type u
  val v : u
  val f : u -> bool
end

module K = struct type t = int end

Figure 3.2: fct.ml

module I = SetFn(K)

Figure 3.3: apply.ml
Separating a functor F and the argument K into different files (figs. 3.2 and 3.3), loading either of them more than once, and applying the functor will cause the types I.t and I0.t to be unequal under applicative semantics because they are either applications of the same syntactic functor to two different modules K (loading K twice gives rise to two different syntactic modules) or different syntactic functors to the same syntactic K.

The above examples also come up in batch compilation. If both functor S and argument module K are defined in a compilation unit F (fig. 3.4), then applicative semantics deems the client module in fig. 3.5 well-typed.

Independently developed compilation units may each use an int set which is derived from independent applications of the generic set functor to an int key structure. The programmer might want these compilation units to exchange set values. Consider the example in fig. 3.6. Unfortunately, applicative functors cannot help in this latter case. Because separate compilation loads the IntKey source twice, I0 and I1 are effectively applied to two different structures from a syntactic perspective.

Within a single compilation unit, the utility of applicative functor semantics seems questionable. If a functor has already been applied to an argument once, there is no need to apply
it again. The result of the original functor application still exists and should be accessible. If the goal is to manage the namespace, then SML’s semantics of structure abbreviations sharing types even opaquely ascribed ones is sufficient.

Another problem with applicative functor semantics is that it is incompatible with type generativity. Consider the following example:

```ocaml
functor F(functor G(X:T):T) =
  struct
    datatype s = S of int
    structure M = G(struct type t = s end)
    type u = M.t
  end
```

The semantics of datatypes is such that every application of F should produce a fresh tycon for s. The functor application of G depends on both the formal functor G and the fresh tycon s. The actual tycon s passed to the functor G is not known until F is applied, thus the dependency certainly cannot be expressed in terms of an applicative functor path.

The other potential application is in distributed memory systems. Multiple computing nodes may each apply a functor to the same argument and then try to exchange values of a type in the result. Such a scenario will likely require runtime types. One argument in favor of applicative functors is in the construction of Okasaki’s bootstrapped heaps [53]. However, even in that case,

In OCaml, the syntax reflects the absence of generativity: there is no functor definition syntax that admits an empty parameter `functor F() = ....`. Because the identity of functor body types depends on a syntactic parameter, each functor application must have one. Applicative functors are also incompatible with shadowing, a standard feature in functional languages. Because its notion of type equality is syntactic, OCaml does not support shadowing of type
declarations in structures. For example, the following does not elaborate in OCaml:

```ocaml
module M = struct type t = A type t = B val x = A end
```

### 3.1.1 Generative Functors

There are implicitly two types of functors, those that have static effects, *generative* functors, and those that do not, *pure* functors. In the higher-order case, pure functors can be fully described using applicative functor-like syntactic extensions. The application of a functor to the same argument twice should give the same result because pure functor arguments are truly referentially transparent. This observation is certainly false when considering generative functors which have static effects.

Type generativity is typically modeled using existential types. Each time an existential type is unpacked, a fresh abstract type is created. Dreyer, Crary, and Harper [16] advocated treating opaque sealing as a computational effect, a runtime fresh type generation. Dreyer [12] reconciles this account of generative types with recursion via a backpatching type semantics.

### Type Sharing

Type sharing is necessary to support type-safe composition of modules. When a functor is parameterized over two modules which each declare an abstract type, the elaborator assumes that the two types are different. Consider the following example:

```ocaml
signature S0 = sig
  type t
  val a : t
end

signature S1 = sig
  type t
  val f : t -> int
end

functor F(structure X:S0 structure Y:S1) = struct
  val y = Y.f(X.a)
end
```
The above functor will not type check because there is no reason to believe \(a \, \text{priori}\) that \(X.t\) and \(Y.t\) are equivalent. \(F\) can certainly be applied to \(X\) and \(Y\) such that \(X.t \neq Y.t\). If \(F\) is applied to \(X\) and \(Y\) such that \(X.t = Y.t\), then this is an instance of the \textit{diamond import problem}. Simply put, when a single component, which may be a type or a structure, is exported in two different structures, it must be reconciled, retaining its identity and thus reflexive equivalence. The ML solution to this problem is type sharing. \textit{Type sharing} can be expressed in four forms. \textbf{sharing type} constraints specifies equivalences on symbolic paths to types. The right hand side of \textbf{sharing type} must consist of only symbolic paths to types.

\begin{verbatim}
functor F(structure X:S0
           structure Y:S1 sharing type X.t = Y.t) = ...

where type, which is derived from the categorical notion of fibrations, defines abstract type specs \textit{post hoc}. The left hand side of the \textbf{where type} should be a signature \(S1\) containing the type to be constrained. The right hand side consists of the name of the type \(t\) in \(S1\) and an arbitrary type expression such as \(X.t\) or \(X.t\) list.

\begin{verbatim}
functor F(structure X:S0
           structure Y:S1 where type t = X.t) = ...

Definitional type specs constrains type specs inside of signatures. Consequently, they only apply if the signature is expanded out in the functor formal parameter. From the perspective of implementation, \textbf{where type} elaborates to definitional type specs by pushing down the \textbf{where type} definition to the relevant type spec.

\begin{verbatim}
functor F(structure X:S0
           structure Y:sig
               type t = X.t
               val f : t -> int
           end) = ...
\end{verbatim}

The type sharing can also be expressed in terms of signatures parameterized on structures. This alternative was considered and rejected during the initial development of the ML module system in favor of \textbf{where type} and \textbf{sharing type} because parameterized signatures require programmers to anticipate all such sharing when writing the signatures.
3.2 First- Versus Second-Class Modules

In most dialects of ML, the module system is second-class, meaning that there is a clear demarcation between the core and module languages. Module expressions cannot depend on the result of core language computations. Core language expressions can only access core language components in modules. They cannot operate on whole modules. This restriction is lifted in module systems with first-class modules, which in practice are useful for supporting plugins. The example for first-class modules often involve some runtime condition that determines the optimal abstract data structure to use, made available as modules, as illustrated by the example given by Dreyer [13]:

```
structure M = if n < 20 then LinkedList else HashTable
```

The above is a kind of optimization that does not require first-class modules in their full generality. Neither, I would argue, does it fit with the main roles of a module system. Because the types in the structures LinkedList and HashTable may be completely different, the static description of M cannot be determined at compile-time. To avoid all the associated issues, language designers have often decided to limit first-class modules by requiring that such module expressions are opaquely ascribed to a single signature, an encoding which is a variation on existential types. Two proposals for first-class modules, Dreyer [13] and Rossberg [61], take this approach.

```
structure M = if n < 20 then LinkedList else HashTable
  :- sig type t type item val insert : t * item -> t ...
end
```
3.3 Recursive Modules

In contrast to the triviality of mutual recursion in most programming languages, recursion at the module level is highly involved for module systems in statically-typed languages. Both the dynamic and static semantics are complicated by the static typing in the module system. Instead of the simple fixed point semantics for recursive module evaluation, the presence of type generativity necessitates alternatives because otherwise datatypes and opaque ascription will have generated new tycons for each recursive call. Even in the absence of datatype declarations and opaque ascriptions, ref cells and assignment in the core language are enough to complicate recursion at the module level. The standard work-around is to adopt Scheme’s backpatching semantics except for the static part of the module.

Backpatching semantics, however, must be made statically safe by ensuring that no recursive variable is used before it is backpatched. Furthermore, recursive modules should be amenable to separate compilation. Dreyer [13] uses the concept of evaluability to enforce safety where a module is evaluable if it does not dereference an unbackpatched recursive variable. The characterization of evaluability is imprecise. The type system based on evaluability cannot determine the safety of some module expressions.

The problem of type inference for recursive modules is complicated. In Dreyer’s language, all recursive variables must be explicitly typed. Dreyer considers extending type inference to recursive variables, but this would require a complicated effect inference technique that produces large and complex types that possibly lack principality in the first place.

3.4 Signature Ascription

Where the core language has type ascriptions, existing module systems support ascribing signatures to module expressions. Ascriptions are even more important in the module system than in the core language because module languages are explicitly typed and some forms of
ascription serve as the primary means for enforcing data abstraction.

Opaque ascription (i.e., sealing) should be construed as a deliberate design choice on the part of the programmer to make a type unique and incomparable. When using opaque ascription, the semantics should be simple. Alternative forms of opaque ascription such as the forms proposed by Dreyer [13] and Govereau [26] that make a type unique in some cases yet simultaneously comparable introduces a new set of issues that complicate the module system.

3.5 Separate Compilation

Early module systems had two main guiding principles. First, modules should be partitioned into interface and implementation. The interface describes the form of a module, the implementation the details and mechanisms. Second, one should be able to compile modules independently from one another with only the aid of interfaces to resolve intermodule dependencies. In languages with simple type systems, reconciling these two goals is usually quite straightforward.

The fundamental tension between separate compilation and an expressive, flexible module system is this: separate compilation requires the language to expose as much as possible in the interface or signature language but the goal of modularity is to keep a distinction between interface and implementation. Reconciling these two goals is a challenge in the ML language because of the presence of static effects. These effects, embodied in type generativity, are computed during the compiler elaboration process. Normally, a signature language should only contain information about the form of the module being described. Inclusion of static effect information would amount to adding information about the implementation. I contend that including static effects would run against the purpose of signatures. Moreover, it would entail significant code duplication and encourage further breakdown of the distinction between interface and implementation.
As Russo [63] noted because ML type information is spread between signature and realization, module clients that depend on the realization cannot be typechecked separately from the module.

Normal ML syntactic signatures cannot even distinguish between functors that contain static effects and those that do not much less the nature of the static effect. For example, a functor body signature may contain a datatype declaration. Naively, an elaborator might consider this an indication that the functor body has a static effect, generating a fresh datatype upon each application. However, this assumption is incorrect because this specification matches both datatype replication declarations, which do not generate a fresh type, *bona fide* generative datatype declarations.

```ml
datatype t = K
structure M : sig datatype t = K end =
struct
datatype t = datatype t (** pure **) end
```

Furthermore, the signature spec could have been `type t` or `datatype t = datatype t`. Consequently, an elaborator cannot handle the special case of pure functors by looking at the functor signature because they are indistinguishable from the signature.

In order to support true separate compilation, the syntactic signature language must be extended with the entire entity calculus. The essence of the entity calculus is a small, simple language to express functor actions including type generativity. However, programming directly in the entity calculus would be an onerous task. One must account for all the potential functor actions when writing functor signatures. It would be much simpler to have the compiler produce entity expressions for describing functor actions as the elaboration semantics in the previous chapters has done.

### 3.5.1 Target Calculi and Type Sharing

ML compilers often use enriched System F_\(\omega\)-like intermediate languages. Unlike the module system, System F_\(\omega\) does not have any native support for type sharing. Naively compiling the
type sharing example in sec. 3.1.1 produces $\Lambda t.\lambda a : t \Lambda s.\lambda f : s \to int.f(a)$, which is ill-typed. There are a number of solutions to this problem.

1. Singleton kinds express the type sharing as a kinding of type $s$: $\Lambda \lambda a : t \Lambda s : \{t\}.\lambda f : s \to int.f(a)$. This technique is used in the TIL/TILT compilers and the Harper-Stone semantics.

2. The $s$ dependency can be abstracted and expressed in terms of System F$\omega$’s type constructors and expressions: $\lambda t.\lambda a : t.\lambda f : (\lambda s.s \to int) f(a)$

3. FLINT preprocesses type sharing by replacing defined type names with their definitions: $\Lambda t.\lambda a : t.\lambda f : t \to int.f(a)$

### 3.5.2 Alternatives

Despite its difficulty, true separate compilation is still desirable. As others have noted, one solution is to distinguish between compilation units and modules. Indeed, OCaml, Moscow ML, and Swasey’s SMLSC take exactly this approach. In OCaml, each separate OCaml source file is a compilation unit which is a kind of quasi-module which must be coupled with an mli interface file. Because there is no compilation unit-level parameterization, the question of higher-order functors does not apply. Moscow ML only considers anopaquely seal module to be a compilation unit.

### 3.5.3 Relationship to Type Classes

As Wehr notes, module encodings in terms of type classes also conflict with separate compilation. Because type classes are ordinarily transparent, they pose lead to type propagation problems similar to module systems.
3.5.4 Comparison to Shao

Shao’s calculus KMC aims to support both separate compilation and full transparency in higher-order modules by means of higher-order type constructors. The calculus is based on Harper-Lillibridge and Leroy’s abstract approach, utilizing existentials for abstract types. The general idea is to factor out all the volatile components to a single higher-order type constructor. Inside of functor signatures, one can use selectors to project out the volatiles from this type constructor. Signatures are then parameterized by this higher-order type constructor.

The Apply functor’s signature would be \( \lambda u_1 : K.\Pi X : (\exists u_2. SIG[u_2]).(SIG[u_1 X]) \). The higher-order type constructor \( u_1 \) plays a role similar to structure realizations and functor realization parameter. However, whereas structure realizations are completely distinct and nonoverlapping with signatures in M, the higher-order tycon is mixed in with syntactic signatures. This affords Shao’s calculus the possibility of separate compilation.

KMC’s main limitation, however, is that it is still not flexible enough to express generative types and abstract types. Though KMC can model opaque sealing via existential types, it still cannot type check programs such as the following:

```plaintext
signature SIG = sig type t val x : t end
funsig FSIG(X: SIG) = SIG
structure S = struct type t = int val x = 1 end
functor APPS (F:FSIG) = F(S)
functor G2(X: SIG) =
  struct
    datatype t = A
    val x = A
  end
```

45
CHAPTER 4  
TYPE SYSTEM

The syntactic type system consists of monotypes, type constructors (tycons), and type expressions (aka types). The monokind \( \Omega \) and arrow kind \( \Omega^n \Rightarrow \Omega \) classify monotypes and tycons respectively. As can be seen from the form of the arrow kind, the syntactic type system does not support higher-kind ed tycons but it does support multi-arity tycons where the \( n \) is a natural number indicating the arity of the tycon. Furthermore, the core language does not support impredicative polymorphism. Universal quantifiers for polymorphism are added in a semantic type system introduced later in this chapter. In the surface type language, the only type expressions that can occur on the right hand side of type definitions are type variables, arrow types, and type applications. Type functions are only introduced by the type definition construct in the core language.

The kind and tycon language follow a significantly simplified version of Dreyer’s language[13]. The tycon language is an impredicative variant of System \( F_\omega \) for simplicity, although the core language does not take advantage of impredicativity because the tycon expressions are an elaborated version of the core language types, which do not contain universal quantifiers at all since the quantifiers are added at the module declaration level. Type expressions \( (T) \) classify core language terms. Monotypes may be of two forms. Type variables \( (\alpha) \) come from a denumerable set. The tycon application form \( (p(C^s)) \) requires the type operator to named by the symbolic path \( p \). By letting the argument of the application be an empty vector, the

\[
\begin{align*}
  p & ::= \, x \mid p.x & \text{symbolic paths} \\
  K & ::= \, \Omega \mid \Omega^n \to \Omega & \text{kinds} \\
  C^s & ::= \, \alpha \mid p(C^s) & \text{monotypes} \\
  C^A & ::= \, \lambda \alpha.C^s & \text{closed tycons} \\
  T & ::= \, \text{typ}(C^s) \mid \forall \alpha.C^s & \text{closed types} \\
  e & ::= \, p \mid \lambda x.e \mid e_1 e_2 & \text{terms}
\end{align*}
\]

Figure 4.1: Surface type system
form can represent all simple types as an application of a nullary type operator to an empty argument vector.

Wrapping an \( n \)-ary \( \lambda \)-abstraction around a monotype produces a tycon. The \( \lambda \) tycon denotes type functions parameterized by tycon variables \( \vec{\alpha} \). The vector of tycon parameters may be empty thus permitting nullary type functions, which represent nullary tycons.

In the syntactic tycon language, \( \lambda \)-abstractions are \( n \)-ary and not curried, so applications must be saturated. Using the \( \lambda \)-abstraction to represent even nullary tycons ensures that the semantics can distinguish definitional tycons, \( i.e., \) they are exactly those wrapped with an outer \( \lambda \)-abstraction. \( C^\lambda \) are the only syntactic tycons, but semantic tycons, described in sec. 4.2, add forms corresponding to datatype tycons. This does not reduce the expressiveness of the tycon language at all. In fact, it is a very close match to the tycon language in ML, considerably closer than more general languages found in previous treatments.

The universal type expression \( (\forall \vec{\alpha}.C^s) \) represents polymorphism. Universal quantifiers only occur at the structure declaration level.

### 4.1 Kind System

The well-kinding judgments (Fig. 4.2) are of the form \( \Gamma, \Delta \vdash C : K \) where \( C \) is \( C^s, C^\lambda \), and \( T \). The static environment \( \Gamma \) is used to interpret symbolic paths in monotype application forms \( p(C^s) \). At this point, one can think of \( \Gamma \) as a finite partial map from symbolic paths to syntactic tycons. In chapter 7, static environments will be formally defined. \( \Gamma(p) \) calculates a tycon, either a defined tycon or a tycon produced by datatype declaration, which is internal, accessible by a type variable. \( \Delta \)s represent the kind environment, which contains all the bound type variables, which by default have kind \( \Omega \). Again, only monokinds are necessary here because the syntactic type system does not support higher-kind tycons. Also noteworthy is that type variables \( \alpha \) can only be of the monokind (rule 4.1).

The tycon application kinding rule (4.2) checks that the operator tycon path is well-
\[ Path = \text{set of all symbolic paths} \]
\[ Tyc_{syn} = \text{set of all closed syntactic tycons} \]
\[ \Delta ::= \emptyset_{knds} \mid \Delta[\alpha] \quad \text{kind environment} \]
\[ \Gamma : Path \rightarrow Tyc_{syn} \quad \text{static environment} \]

\[ \Gamma, \Delta \vdash C^{s} : \Omega \]

\[ \alpha \in \Delta \]
\[ \Gamma, \Delta \vdash \alpha : \Omega \] (4.1)

\[ \Gamma, \Delta \vdash \Gamma(p) : \Omega^{n} \Rightarrow \Omega \quad |\vec{C}^{s}| = n \quad \Gamma, \Delta \vdash C_{i}^{s} : \Omega \quad \forall i \in [1, n] \]
\[ \Gamma, \Delta \vdash p(\vec{C}^{s}) : \Omega \] (4.2)

\[ \Gamma, \Delta \vdash C^{\lambda} : \Omega^{n} \rightarrow \Omega \]

\[ \Gamma, \Delta[\alpha_{1}] \ldots \alpha_{n} \vdash C^{s} : \Omega \]
\[ \Gamma, \Delta \vdash \lambda \vec{\alpha}.C^{s} : \Omega^{n} \Rightarrow \Omega \] (4.3)

\[ \Gamma \vdash T : \Omega \]

\[ \Gamma, \emptyset_{knds} \vdash C^{s} : \Omega \]
\[ \Gamma \vdash \text{typ}(C^{s}) : \Omega \] (4.4)

\[ \Gamma, [\alpha_{1}] \ldots [\alpha_{n}] \vdash C^{s} : \Omega \]
\[ \Gamma \vdash \forall \vec{\alpha}.C^{s} : \Omega \] (4.5)

Figure 4.2: Well-kindng of syntactic tycons
kinded, the correct number of arguments is supplied and that each of the arguments is well-kindled with kind Ω. The notation \( C^s_i \) denotes the \( i \)th element in the \( C^s \) vector. The rest of the rules ensure that the types and tycons are closed with respect to type variables (i.e., all type variables are in \( \Delta \)).

### 4.2 Semantic Type System

The foregoing type system is purely syntactic. There is no semantics for evaluating the type expressions and tycons. Syntactic types cannot be evaluated directly due to the presence of the uninterpreted symbolic paths. Fig. 4.3 gives a *semantic* type language that replaces the symbolic paths in the tycon application form with tycons. Semantic monotypes, tycons, and type expressions are distinguished from their syntactic counterparts by the Fraktur script (\( \mathcal{C} \)). The semantic type language is first-order and simply-kindled. The Tycs set is the set of all tycon symbols \( \tau^n \), distinct from syntactic symbols. Each such tycon has an assigned arity \( n \). The elements in Tycs will be called *atomic tycons*. All primitive tycons such as int, bool, and arrow fall into the Tycs set and all generated datatype tycons will be elements of Tycs.

The semantic type system consists of semantic monotypes (\( C^s \)), semantic tycons (\( C^\lambda \)), and semantic type expressions (\( \mathcal{T} \)) each of which corresponds to the respective syntactic type system version. Semantic monotypes differ from syntactic monotypes in that semantic tycons replace symbolic paths in the application form. Semantic tycons consist of both a \( \lambda \) form that represents definitional tycons and a \( \tau^n \) form, a concrete, atomic tycon from Tycs,
\[ \Delta \vdash C_s : \Omega \]

\[ \begin{align*}
\alpha & \in \Delta \\
\Delta & \vdash \alpha : \Omega 
\end{align*} \quad (4.6) \]

\[ \begin{align*}
\Delta \vdash C^\lambda : \Omega^n \Rightarrow \Omega & \quad |\vec{C}s| = n \quad \Delta \vdash C_s^i : \Omega \ \forall i \in [1, n] \\
\Delta & \vdash C^\lambda(\vec{C}s) : \Omega 
\end{align*} \quad (4.7) \]

\[ \vdash C^\lambda : \Omega^n \rightarrow \Omega \]

\[ \begin{align*}
[\alpha_1] \ldots [\alpha_n] & \vdash C^s : \Omega \\
\vdash \lambda \vec{\alpha}. C^s : \Omega^n \rightarrow \Omega 
\end{align*} \quad (4.8) \]

\[ \vdash \tau^n : \Omega^n \rightarrow \Omega \quad (4.9) \]

\[ \vdash T : \Omega \]

\[ \begin{align*}
\emptyset_{knds} & \vdash C^s : \Omega \\
\vdash \text{typ}(C^s) : \Omega 
\end{align*} \quad (4.10) \]

\[ \begin{align*}
[\alpha_1] \ldots [\alpha_n] & \vdash C^s : \Omega \\
\vdash \forall \vec{\alpha}. C^s : \Omega 
\end{align*} \quad (4.11) \]

---

**Figure 4.4: Well-kindig of semantic tycons**

which is intended to represent tycons generated by datatype declarations, opaque ascription, and functor applications. The arity of well-formed tycons can always be calculated directly, \( |\tau^n| = n \) and \( |\lambda \vec{\alpha}. C^s| \).

Fig. 4.4 gives a kind system for semantic monotypes, tycons, and type expressions. This kind system differs from the the previous one in fig. 4.2 in that the application rule 4.7 no longer needs to look up a static environment for a symbolic path, because there are no symbolic paths. The simplifies the kind system in that the static environment can be dropped. To handle atomic tycons, the kind system also adds rule 4.9 that gives it the arrow kind \( \Omega^n \Rightarrow \Omega \) to match its arity. Thus, the role of this kind system is solely to verify that arities match up and that the types are closed with respect to type variables.
4.2.1 Notation

Let $C^\lambda$ be a semantic tycon. $|C^\lambda|$ refers to the arity of the tycon. If the tycon is an atomic tycon $\tau^n$, then $|\tau^n| = n$. If the tycon is a $\lambda\vec{\alpha}.C^s$, then the arity is the length of the $\vec{\alpha}$. The $|\cdot|$ notation is also carried over in the obvious way to relativized tycons. $C^{nf}_1 \equiv_{\alpha} C^{nf}_2$ is the equivalence of two semantic monotypes module $\alpha$-renaming of type variables, i.e. the two monotypes are isomorphic up to $\alpha$-renaming of type variables.

4.2.2 Evaluation

Semantic type expressions and monotypes can be evaluated by a $\beta$-reduction rule in Fig. 4.5 to a normal form $C^{nf}$. Note that the normal form of atomic tycons with 0-arity is $\tau^0()$.

The evaluation rule is the typical big-step call-by-value $\beta$-reduction modified to accept a vector of arguments. Since the semantic type language is analogous to a simply-typed $\lambda$-calculus (actually a first-order simply-typed $\lambda$-calculus enriched with constants), the language is strongly normalizing (lem. 18).

Lemma 1 (Monotypes are strongly normalizing)

If $C^s \Downarrow_{mt} C^s_1$ and $C^s \Downarrow_{mt} C^s_2$, then $C^s_1 = C^s_2$ and $C^s_1$ is a $C^{nf}$.

Not only are the types strongly-normalizing, the semantic type language also has preservation and progress properties.

Lemma 2 (Kind Preservation)

If $\emptyset_{knds} \vdash C^s : \Omega$ and $C^s \Downarrow_{mt} C^{nf}$, then $\emptyset_{knds} \vdash C^{nf} : \Omega$. 

\[ \frac{C^s \Downarrow_{mt} C^{nf}_1, C^s\{C^{nf}_1/\vec{\alpha}\} \Downarrow_{mt} C^{nf}_2}{(\lambda\vec{\alpha}.C^s)(\vec{C}^s) \Downarrow_{mt} C^{nf}_2} \quad (4.12) \]
Lemma 3 (Progress)

If $\emptyset \vdash e^s : \Omega$, then $e^s \downarrow_{mt} e^{nf}$.

The connection between the syntactic and semantic type systems will be made clear in chapter 7.
CHAPTER 5
SURFACE LANGUAGE

5.1 Core Language

The surface language (Fig. 5.1) closely follows SML/NJ’s syntax with a few exceptions. There
are three forms of core level declarations: value, type definition, and datatype. For type
definitions, the type parameters are written in an explicit λ-abstraction form \texttt{type } t = \lambda \vec{\alpha}.C^s
instead of \texttt{type } \vec{\alpha} t = C^s. Definitional type specs use a similar notation. For simplicity, the
data constructor part of datatype declarations is omitted. The interesting behavior is the
datatype declaration’s generation of a fresh atomic tycon.

The core language is a simple one comprised of an implicitly typed expression language
(e) and a simple declaration language (d\textsuperscript{C}). There are three forms of declarations in the core
language, value declarations, type definitions, and datatype declaration. For simplicity’s
sake, datatype declarations only declare the name and parameters (arity) of a generative
 tycon.

5.2 Signatures

A signature is a tree of static information for a structure. This object is a hierarchy because
it must reflect the hierarchical nesting of structures. The hierarchy can be considered a tree
where the internal nodes are structure specifications (\textit{i.e.}, a signature for a named structure)
and the leaves are static entities for tycons and functors.

Signatures are defined as a sequence of comma separated specs. Value specs use the
notation \texttt{val x : \forall \vec{\alpha}.C^s} instead of \texttt{val} \vec{\alpha} x = C^s. Each sequence is terminated by the empty
spec \emptyset_{specs}. Signature expressions include a variable form, base signature \texttt{sig spec end}, and
a signature expression modified by a where type clause. Signature declarations are always
at top level.

53
In signatures, there are two kinds of tycon specifications, definitional and open. *Definitional tycon specs* (abbreviated as *definitional specs*) constrain tycons that can be matched. *Open tycon specs* (abbreviated as *open specs*) only declare the name and arity of a tycon that must be present in a matching structure.

```plaintext
type ('a, 'b) s = 'a list * 'b (* definitional *)
type ('a,'b) t (* open *)
```

Definitional specs are intended to match type definitions in structures. Open specs can match type definitions, datatype declarations, and abstract types (i.e., a type component made abstract by opaque ascription) in structures.

Signature expressions may be the base form that contains specs. The where type form of signature expressions is a generalization of SML’s mechanisms. The where type form adds the type definitions on the right hand side to the structure spec (signature expression) identified on the left hand side.

Since signature expressions support the base form of signatures and these expressions are found in signature specs, module declarations, and top level declarations, this amounts to the possibility of inline base signatures throughout the language. The presence of inline signatures must be accounted for during elaboration.
5.3 Module Calculus

Structure expressions can be a path $p$, a base structure, a functor application (where the functor occurs only as a symbolic path $p$), transparent signature ascription, and opaque signature ascription. A module declaration is a sequence of structure, functor, and core declarations. $\circ$ denotes the empty structure declaration that terminates every sequence of structure declarations. At the top level, in addition to module declarations, one may also have signature declarations. The top level declaration sequence is terminated by an empty top level declaration $\circ$.

The semantics of transparent and opaque ascription follow that of Standard ML. Unlike some of the more recent module systems, Standard ML’s signature ascription is always a coercion on the structure to the form specified in the signature. In the transparent case, this amounts to dropping any value bindings or type declarations omitted in the signature. However, when a signature’s open spec is matched against a structure’s type definition, the resulting coerced structure will reveal that type definition. In the opaque case, this kind of type definitions is occluded.

5.4 Primary and Secondary Components

The form of a functor argument is constrained by the functor parameter signature possibly modified by a where type definition. In the parameter signature, there can be structure specifications, formal functor specifications, structure/type sharing constraints, and the two classes of type specifications. The open and definitional specs in functor formal parameter signatures are worth special mention. The tycon names declared by these specs may occur in the functor body. When typechecking these occurrences, a free instantiation (i.e., dummy instantiation) of the tycon is used. SML/NJ also supports a rich notion of symmetric type sharing constraints, that will be deferred for future work. Where type constraints can turn
open specs into definitional ones when pushed down. Open specs that remain open specs after the resolution of all sharing and where type constraints correspond to these free instantiations. These free instantiations are called formal tycons. These are exactly the flexible tycons mentioned in Shao [68].

In the presence of type sharing constraints, which induce an equivalence class of formal tycons, each equivalence class has a canonical representative called a primary tycon. Without type sharing constraints, all formal tycons are primary. These primary tycons are the essential components that must be kept to maintain the semantics of functor application (i.e., the type application associated with the functor application). References to all other members of the equivalence class should be redirected to the associated primary type component. The remaining type components (from definitional specs) are secondary and therefore should be fully derivable from the primary tycons and externally defined types. Secondary types do not have to be explicitly represented in the parameter signature because all occurrences of these secondary types can be expanded out according to their definitions.

functor \( F(X:\text{sig type } s \text{ type } t \text{ type } u = s \ast t \text{ sharing type } t = s \text{ end}) = \ldots \)

In the above example, \( s \) can be primary, representative for the equivalence class containing both \( s \) and \( t \), and \( u \) is secondary. Primary tycons are the ones that must be instantiated each time a functor is applied. The primary tycons must then be substituted into the secondary tycons or, alternatively, looked up in an environment described in chapter 6.

## 5.5 Nonvolatile and Volatile Type Constructors

In the module system, tycons can be classified as either nonvolatile or volatile. Nonvolatile tycons are atomic tycons in the initial static and entity environments. Open type specs in signatures, datatype declarations, and tycons defined relative to former two give rise to volatile tycons, i.e., they may be instantiated to a particular tycon by functor application or signature matching. Moreover, volatile tycons can be instantiated multiple times through
different functor applications or signature matching. Although volatile tycons overlap with the notion of abstract types, they are not the same. The defining characteristic of volatile tycons is that their actual instantiation will be supplied at a later point and they can have multiple instantiations within a program. The instantiation used changes depending on the context, thus the tycon is volatile. In cases such as transparent signature ascription, a volatile tycon may be perfectly transparent.

Volatile types themselves can be classified as primary (open tycon specs in functor formal parameters and signatures) and secondary (tycons defined relative to primary volatile tycons).

```ocaml
functor F(X: sig
    type a (* primary, volatile *)
    type b = int (* secondary, nonvolatile *)
  structure M0 :
    sig
      type c (* primary, volatile *)
      type d = a (* secondary, volatile *)
    end
  end
end

structure M1 :
  sig
    type v (* spec, not a tycon *)
  end = struct
    type v = X.M0.c * int (* secondary, volatile *)
  end
end
```

The above example illustrates some of the complexities of volatile types. Types X.a and X.M0.c are obviously volatile. Definitional types X.M0.d, u, and M1.v are also volatile because they are defined in terms of volatile types.
CHAPTER 6

ENTITY CALCULUS

In this module system, I will use an entity calculus, a refined version of the static representation of functor actions and static module content found in the SML/NJ compiler. The entity calculus provides a natural representation of the static content of modules, which I call static entities. The term entity refers to an extended form of type constructors ($\mathbb{C}^{\lambda}$) and a static description of objects that may include type constructors such as structures and functors. In the module system, syntactic signatures alone are insufficient for expressing functor actions. The entity calculus is a small language for expression exactly that, the functor actions for a functor.

Fig. 6.1 describes the entity calculus, comprising of entity expressions $\varphi$, entity declarations $\eta$, and static entities $\upsilon$. The static entities are the entity calculus analogue to values in the lambda calculus. The representation relies on a denumerable set of variables, EntityVars, which represents unique names that can be thought of as the new variable names resulting from alpha-conversion.

Entity environments $\Upsilon$ are sequences of bindings of an entity variable $\rho$ to a static entity $\upsilon$. The sequence is important because the semantics needs a deterministic way to search through the environment for atomic tycons. The role of entity environments is two-fold. They give the current instantiation of a relativized tycon, thus enabling translation of a relativized tycon to a specific semantic tycon. Furthermore, they permit the translation of a semantic tycon back to an entity path, a canonical relativized tycon.

There are four forms of static entities, corresponding to structure, functor, atomic tycons, and definitional (derived) tycons. Structure and functor entities are enclosed in $\langle \rangle$ brackets. A structure entity $\langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle$ has a local entity environment $\Upsilon^{lcl}$ and a closure environment

---

1. Entity calculus tycons differ from syntactic types only in that the tycons may have been relativized, a process described in a later section, and prefixed polymorphic (universal) quantifiers.
\[ \rho \in \text{EntityVars} \quad \text{entity variables} \]
\[ \Upsilon ::= \emptyset_{ee} \mid \Upsilon[\rho \mapsto \upsilon] \quad \text{entity environment} \]
\[ R ::= \langle \Upsilon_{lcl}, \Upsilon_{clo} \rangle \quad \text{structure entity} \]
\[ \psi ::= \langle \lambda \rho. \varphi; \Upsilon \rangle \mid \langle \lambda \rho. \Sigma; \Upsilon \rangle \quad \text{functor entity} \]
\[ \upsilon ::= \psi \mid R \mid \tau^n \mid \mathcal{E}^\lambda \quad \text{static entity} \]

Tycon entity expression
\[ \zeta ::= \vec{\rho} \mid \text{new}(n) \mid \mathcal{C}^\lambda \]

Structure entity expression
\[ \varphi ::= \vec{\rho} \quad \text{entity path} \]
\[ \mid \langle \eta \rangle \quad \text{entity declaration} \]
\[ \mid \vec{\rho}(\varphi) \quad \text{functor application} \]
\[ \mid \text{let } \eta \text{ in } \varphi \quad \text{local definition} \]
\[ \theta ::= \vec{\rho} \mid \lambda \rho. \varphi \mid \lambda \rho. \Sigma \quad \text{functor entity expression} \]
\[ \eta ::= \bullet \mid [\rho =_{tyc} \zeta] \eta \mid [\rho =_{str} \varphi] \eta \mid [\rho =_{fct} \theta] \eta \quad \text{entity declarations} \]

Figure 6.1: Static entities and entity expressions

\[ \Upsilon_{clo} \] to close an associated semantic signature with respect to any entity variable not defined in \[ \Upsilon_{lcl} \], \textit{i.e.,} the nonlocal entity variables.

Consider the above example. The semantic signature \( S' \) has a free occurrence of the entity variable for \( t \rho_t \). The entity variable \( \rho_t \) is not in the domain of the entity environment for \( S' \). In fact, the entity environment for \( S' \) is empty because \( S' \) does not contain any static entities. Still, the structure entity for \( S' \) must be self-contained. Therefore it must close the signature for \( S' \) with a closure entity environment which is a copy of the entity environment for its enclosing structure. A structure entity encodes only the volatile type information in structure, \textit{i.e.,} the part that is re-instantiated with signature matching and functor application.
There are two forms of functor entity. The standard form $\left< \lambda \rho. \varphi ; \Upsilon \right>$ encodes the functor action as an entity function $\lambda \rho. \varphi$ together with a closure entity environment $\Upsilon$. Applying an entity function results in a structure entity. The alternate form $\left< \lambda \rho. \Sigma ; \Upsilon \right>$ represent functors in formal functor parameters where the only information known is the semantic signature $\Sigma$ of the functor result, described in chapter 7 in fig. 7.2. Semantic signatures are syntactic signatures where each of the static components (tycons, structures, and functors) are decorated with entity variables. They are formally defined in chapter 7. A semantic signature may mention entities by their entity variables (called relativized tycons), hence it must closed by an entity environment $\Upsilon$. Relativization is formally defined in sec. 6.4. Atomic tycons $\tau^n$ and definitional tycons $\mathcal{C}^\lambda$ (defined in sec. 6.4.1) are the tycon entities. Entity environments may contain definitional tycons because of signature matching, a subject thoroughly discussed in sec. 7.12.

Static entities are the values computed by the entity calculus. Tycon entity expressions evaluate to tycon entities. Previously declared tycons and formal tycons from functor parameters are represented by their entity path, $\vec{\rho}$. They can also be a new($n$) or a relativized tycon $\mathcal{C}^\lambda$ (defined in sec. 6.4) that refers to a tycon whose instantiation is in an entity environment. The new($n$) form produces a new atomic tycon $\tau^n$ when evaluated where $n$ specifies the arity of the desired tycon. The new($n$) expression is the key to representing type generativity in the entity calculus. Datatypes and other generative tycons are represented by the form new($n$) which when evaluated will generate a fresh type constructor unequal to all those generated before it.

Structure entity expressions and functor entity expressions evaluate to structure entities and functor entities respectively. The entity path form in both structure and functor entity expressions is a reference to an entity bound in a previous entity declaration. Structure entity expressions also consist of entity declarations enclosed by $\langle \rangle$ corresponding to the struct ... end base structure form, functor application expressions, and let expressions. The
let expression is used to express coercions induced by signature matching (see sec. 7.12). In addition to an entity path, functor entity expressions may be entity lambda abstractions or formal functors of the form \( \lambda \rho. \Sigma \).

Entity declarations bind an entity variable to tycon entity expressions, structure entity expressions, and functor entity expressions. The entity calculus has a static compile-time semantics. The elaborator evaluates entity expressions to static entities as necessary during elaboration. The judgments for the entity expression evaluation are outlined below:

\[
\begin{align*}
\Upsilon & \vdash \varphi \Downarrow_{\text{str}} R & \text{structure entity expression evaluation} \\
\Upsilon & \vdash \eta \Downarrow_{\text{decl}} \Upsilon' & \text{entity declaration evaluation} \\
\Upsilon & \vdash \theta \Downarrow_{\text{fct}} \psi & \text{functor entity expression evaluation} \\
\Upsilon & \text{is an entity environment. } \varphi & \text{is the resulting entity expression. } \psi & \text{is a resulting functor entity. } R & \text{is the entity environment obtained by evaluating the structure entity expression. } \\
\Upsilon' & \text{is the entity environment obtained by evaluating the entity declarations } \eta. & \text{Each binding in } \Upsilon' & \text{corresponds to an entity declaration in } \eta, & \text{thus } \Upsilon \not\subseteq \Upsilon'.
\end{align*}
\]

### 6.1 Entity declarations

The evaluation rules for entity declarations must accumulate the entity environment across declarations in order to support occurrences of entities in value specs within inline open signatures.

```plaintext
datatype d = A
structure M : sig type t val x : d * t end =
struct
    datatype t = B
    val x = (A, B)
end
```

Observe that the type name \( d \) is a nonlocal name that must be defined in a prior entity declaration. The clause entity environment for \( M \) must contain a binding for \( d \).
6.2 Notation

I indicate extension of entity environments by juxtaposition, \( \Upsilon[\rho \mapsto v] \). This notation should never cause shadowing of entity variable bindings because the convention is that \( \rho \notin \text{dom}(\Upsilon) \). Entity environment concatenation of two entity environments \( \Upsilon \) and \( \Upsilon' \) is represented by juxtaposition, \( \Upsilon \Upsilon' \). The notation for entity environment lookup is overloaded. \( \Upsilon(\rho) \) looks up the static entity corresponding to \( \rho \). The notation \( \Upsilon(\rho_0 \vec{\rho}) \) looks up \( \rho_0 \) in \( \Upsilon \) for a structure entity \( R = \langle \Upsilon^{\text{lcl}}, \Upsilon^{\text{clo}} \rangle \) and then recursively computes \( \Upsilon'(\vec{\rho}) \) such that \( \Upsilon' = \Upsilon^{\text{clo}} \Upsilon^{\text{lcl}} \). Let \( \text{EV}(\cdot) \) denote the set of all entity variables in \( \cdot \). The notation \( \text{dom}(\Upsilon) \) denotes the set of entity variables in the domain of all mappings in \( \Upsilon \), including the nested ones.

Let \( R \) be a structure entity \( \langle \Upsilon^{\text{lcl}}, \Upsilon^{\text{clo}} \rangle \) and \( \vec{\rho} \) be an entity path defined in either \( \Upsilon^{\text{lcl}} \) or \( \Upsilon^{\text{clo}} \). Then \( R(\vec{\rho}) \) is defined such that \( R(\vec{\rho}) = \Upsilon^{\text{clo}} \Upsilon^{\text{lcl}}(\vec{\rho}) \).

\( \Upsilon_1 \subseteq \Upsilon_2 \) indicates all the bindings in \( \Upsilon_1 \) are in \( \Upsilon_2 \). The notation \( R \subseteq \Upsilon \) means \( \Upsilon^{\text{clo}} \Upsilon^{\text{lcl}} \subseteq \Upsilon \) where \( R = \langle \Upsilon^{\text{lcl}}, \Upsilon^{\text{clo}} \rangle \). Similarly, \( \Upsilon \subseteq R \) indicates \( \Upsilon \subseteq \Upsilon^{\text{clo}} \Upsilon^{\text{lcl}} \). The convention that that if \( R \) is used in a place where an entity environment is expected, then \( R \) denotes \( \Upsilon^{\text{clo}} \Upsilon^{\text{lcl}} \) such that \( R = \langle \Upsilon^{\text{lcl}}, \Upsilon^{\text{clo}} \rangle \).

6.3 Entity Evaluation Semantics

Entity declarations evaluate to entity environments. The module system evaluates entity expressions when elaborating functor application (rule 7.33). Each entity expression represents the latent static computation in a structure or functor expression. Once evaluated, the resultant structure entity provides a precise and complete (closed) description of the entities inside a structure. Evaluation requires an entity environment as context to interpret entity paths in the entity expressions and declarations.

Fig. 6.2 lists the rules for evaluating entity expressions. The key rules are the ones for application (6.3). There are two, one for regular functor application and the other for formal
functor application. The functor entity expression is evaluated to an entity function. The argument structure entity expression is evaluated to an entity environment. The body of the entity function is then evaluated in an entity environment extended with the closure environment and the formal parameter entity variable mapped to the argument structure entity environment. Rule 6.4 evaluates the functor entity expression to a functor entity corresponding to the formal functor case and also the structure entity expression to the structure entity. The semantic signature Σ (i.e. the formal functor body signature) must be instantiated (a process that is covered in sec. 7.8). The rule then forms a structure entity using this instantiation and the extended entity environment as a closure. The instantiation part of the closure will correspond to the actual functor closure when the actual functor is known.

Entity declarations must be evaluated in sequence. Fig. 6.3 formally defines how to evaluate entity declarations. Each tycon, structure, and functor entity declaration evaluates
\[ \Upsilon \vdash \eta \downarrow_{\text{decl}} \Upsilon' \]

\[ \Upsilon \vdash \cdot \downarrow_{\text{decl}} \emptyset \]  \hspace{2cm} (6.6)  

\[ \Upsilon \vdash \varphi \downarrow_{\text{str}} R \quad \Upsilon[\rho \mapsto R] \vdash \eta \downarrow_{\text{decl}} \Upsilon' \]

\[ \Upsilon \vdash \rho =_{\text{str}} \varphi, \eta \downarrow_{\text{decl}} [\rho \mapsto R]\Upsilon' \]  \hspace{2cm} (6.7)  

\[ \Upsilon \vdash \theta \downarrow_{\text{fct}} \psi \quad \Upsilon[\rho \mapsto \psi] \vdash \eta \downarrow_{\text{decl}} \Upsilon' \]

\[ \Upsilon \vdash \rho =_{\text{fct}} \theta, \eta \downarrow_{\text{decl}} [\rho \mapsto \psi]\Upsilon' \]  \hspace{2cm} (6.8)  

\[ \Upsilon[\rho \mapsto \tau^n] \vdash \eta \downarrow_{\text{decl}} \Upsilon' \quad (\tau \text{ is fresh in } \Upsilon) \]

\[ \Upsilon \vdash \rho =_{\text{tyc}} \text{new}(n), \eta \downarrow_{\text{decl}} [\rho \mapsto \tau^n]\Upsilon' \]  \hspace{2cm} (6.9)  

\[ \Upsilon[\rho \mapsto C\lambda] \vdash \eta \downarrow_{\text{decl}} \Upsilon' \]

\[ \Upsilon \vdash [\rho =_{\text{def}} C\lambda], \eta \downarrow_{\text{decl}} [\rho \mapsto C\lambda]\Upsilon' \]  \hspace{2cm} (6.10)  

\[ \Upsilon[\rho \mapsto \Upsilon(\bar{\rho})] \vdash \eta \downarrow_{\text{decl}} \Upsilon' \]

\[ \Upsilon \vdash [\rho =_{\text{def}} \bar{\rho}], \eta \downarrow_{\text{decl}} [\rho \mapsto \Upsilon(\bar{\rho})]\Upsilon' \]  \hspace{2cm} (6.11)  

Figure 6.3: Entity declaration semantics

the respective tycon, structure entity expression, and functor entity expression to the corresponding entity. The subsequent entity declarations are evaluated in the context of an entity environment extended with a mapping for that entity. The entity environment context only has bindings for the entity variables bound in the entity declarations because only entity expression evaluation will provide the correct closure.

Rule 6.6 ensures that the resultant entity environment is cumulative. Rules 6.7 and 6.8 evaluate structure and functor entity declarations in the expected way. Rule 6.9 is the key rule that generates fresh atomic tycons with the appropriate arity \(n\). When the \(\rho =_{\text{tyc}} \text{new}(n)\) entity declaration is embedded in the body of a functor entity, it produces a fresh atomic tycon for \(\rho\) each time the functor entity is applied, \textit{i.e.} also when this entity declaration is evaluated. This models exactly the behavior of generative datatype declarations under true higher-order semantics. Rule 6.10 yields a corresponding type definition entity environment.
binding. Rule 6.11 looks up an entity path (constructed during signature matching coercion rule 7.46) for the corresponding entity mapping.

Fig. 6.4 gives the rules for evaluating a functor entity expression. Rule 6.12 looks up entity paths in the usual way. The entity function is formed by combining an entity \( \lambda \)-abstraction with the current entity environment to form a closure (rule 6.13). Rule 6.14 similarly forms a closure for formal functors.

**Lemma 4 (Entity evaluation terminates)**

The following evaluation relations terminate (have a finite derivation tree):

1. \( \Upsilon \vdash \varphi \downarrow_{str} R \)

2. \( \Upsilon \vdash \eta \downarrow_{decl} \Upsilon' \)

3. \( \Upsilon \vdash \theta \downarrow_{fct} \psi \)

**Lemma 5**

1. If \( \Upsilon \vdash \varphi \downarrow_{str} R \), then \( \Upsilon \subseteq R \).

2. If \( \Upsilon \vdash \theta \downarrow_{fct} \langle \theta ; \Upsilon' \rangle \), then \( \Upsilon \subseteq \Upsilon' \).
6.4 Relativization

Tycon names have a scoping policy much like other program identifiers. Tycon names can be shadowed by later tycon declarations. A tycon name may refer to a local tycon that is declared in the same semantic signature or structure entity, or to a nonlocal one that is declared as a functor parameter or in another structure. Although signature declarations occur only at the top level, anonymous signature expressions may be embedded as functor parameter signatures and signature ascriptions. When signatures occur inline, its type specifications may refer to both local and nonlocal tycons.

Some tycon names may refer to volatile tycons which may be instantiated multiple times possibly to new generative tycon. Each time a functor is applied or a structure is ascribed a signature, the tycon name must be interpreted according to a potentially different entity environment. When elaboration reaches these tycons we need a way to map them back to the corresponding entity path such that we reference the correct “up-to-date” volatile tycon. The necessity of this is illustrated in fig. 6.5. Semantic tycons which live in static environments and entity environments have to be relativized when put into semantic signatures and entity expressions. When the tycon is needed for typechecking, the tycon must be instantiated with respect to an entity environment. This is the primary purpose of relativization, to give an entity path that can be interpreted according to different entity environments. Secondarily, relativization replaces fragile symbolic names with robust entity paths to the entity environment mapping the entity paths.

6.4.1 Relativized Type System

The relativized type system \((C^s, C^\lambda, T)\) replaces all occurrences of atomic tycons with entity paths. The purpose of relativization and this type system is described in section 5.5. Three classes of tycons have been introduced: syntactic, semantic, and relativized. The
semantics progressively translates syntactic tycons to the latter two. The module system only evaluates semantic tycons. Such tycons exist in the static and entity environments. In contrast, relativized tycons are never reduced directly. Instead, they must be instantiated into a semantic tycon for reduction to take place as in fig. 6.5. Relativized tycons exist in semantic signatures and static environment by way of embedded semantic signatures.

Relativized monotypes, tycons, and types must also be well-kinded as defined in Fig. 6.7. The main difference between this well-kinding judgment and the other well-kinding judgments is that rule 6.16 uses the entity environment to obtain a well-kinded semantic tycon. The rest of the semantics is standard.

The rules in Figs. 6.8 and 6.9 define monotype and tycon equivalences and type expressions equivalences respectively. The module system semantics will require a notion of relativized monotype equivalence and relativized tycon equivalence, which are given by rules 6.20 and 6.21 respectively. Monotypes and tycons are equivalent under the interpretation of an
\[\alpha \in \Delta \]
\[\Upsilon, \Delta \vdash \alpha :: \Omega \quad (6.15)\]

\[\Upsilon, \Delta \vdash \Upsilon(\vec{\rho}) :: \Omega^n \to \Omega \quad \Upsilon, \Delta \vdash C_i^s \forall C_i^s \in \overline{C}^s\]
\[\Upsilon, \Delta \vdash \rho(\overline{C}^s) :: \Omega \quad (6.16)\]

\[n = |\vec{\alpha}| \quad \alpha_i \in \vec{\alpha} \quad i \in [1, n] \quad \Upsilon, [\alpha_1] \ldots [\alpha_n] \vdash C^s :: \Omega\]
\[\Upsilon \vdash \lambda \vec{\alpha}. C^s :: \Omega^n \to \Omega \quad (6.17)\]

\[\Upsilon, \emptyset_{knd} \vdash C^s :: \Omega \]
\[\Upsilon \vdash \text{typ}(C^s) :: \Omega \quad (6.18)\]

\[n = |\vec{\alpha}| \quad \alpha_i \in \vec{\alpha} \quad i \in [1, n] \quad \Upsilon, [\alpha_1] \ldots [\alpha_n] \vdash C^s :: \Omega\]
\[\Upsilon \vdash \forall \vec{\alpha}. C^s :: \Omega \quad (6.19)\]

Figure 6.7: Well-ranging of relativized monotypes, tycons, and types

\[\Upsilon(C_1^s) \psi_{mt} C_{1}^{nf} \quad \Upsilon(C_2^s) \psi_{mt} C_{2}^{nf} \quad C_{1}^{nf} \equiv_{\alpha} C_{2}^{nf}\]
\[\Upsilon \vdash C_1^s \equiv C_2^s \quad (6.20)\]

\[|\vec{\alpha}| = |\vec{\beta}| \quad \Upsilon \vdash C_i^s \equiv \{\alpha/\beta\} C_2^s\]
\[\Upsilon \vdash \lambda \vec{\alpha}. C_1^s \equiv \lambda \vec{\beta}. C_2^s \quad (6.21)\]

Figure 6.8: Monotype and type equivalence

\[\Upsilon \vdash T_1 \equiv T_2\]

\[\Upsilon \vdash C_1^s \equiv C_2^s\]
\[\Upsilon \vdash \text{typ}(C_1^s) \equiv \text{typ}(C_2^s) \quad (6.22)\]

\[|\vec{\alpha}| = |\vec{\beta}| \quad \Upsilon \vdash C_i^s \equiv \{\alpha/\beta\} C_2^s\]
\[\Upsilon \vdash \forall \vec{\alpha}. C_1^s \equiv \forall \vec{\beta}. C_2^s \quad (6.23)\]

Figure 6.9: Type expression equivalence
entity environment when they evaluate to equivalent normal form semantic tycons modulo α-renaming. Rules 6.22 and 6.23 describe equivalence for type expressions.

6.4.2 Role of Relativization

Relativization plays a role in signature extraction (discussed in sec. 7.11):

1. Value bindings in the static environment must be relativized.

2. Definitional type binding in the static environment must be relativized in the inferred signature.

```plaintext
sig
  structure M : sig type t end
  type u = M.t
end
```

The type definition for u must be relativized because it refers not to a concrete type, but a type defined relative to a matching structure's realization.

Consider the following example:

```plaintext
structure A =
  struct
    structure B = struct datatype u end
    functor F(X : sig type t end) =
      struct val a : X.t * B.u = ... end : sig val a : X.t * B.u end
  end
end
```

The following are the semantic signatures for the example:

- \( B \mapsto (\rho_B, \{u \mapsto (\rho_u, 0)\}) \)
- \( X \mapsto (\rho_X, \{t \mapsto (\rho_t, 0)\}) \)

In the type for a, the tycon for X.t must be relativized to \( \rho_X \rho_t \) and for B.u to \( \rho_B \rho_u \).

When inferring the signature from the body of F, we have the formal tycon X.t in the static environment. When inferring the signature from F’s body, there are two static en-
environments, the static environment produced by elaborating F’s body and the contextual static environment when entering into the body.

Static environment for F’s body: \([a \mapsto \tau^0_t \ast \tau^0_u]\)

Static environment for the context: \([B \mapsto (\rho_B, \{u \mapsto (\rho_u, \tau^0_u); R_B\})]

\([X \mapsto (\rho_X, t \mapsto (\rho_t, \tau^0_t); R_X) ]\)

\(R_B\) and \(R_X\) are the structure entities for \(B\) and \(X\) respectively.

```plaintext
structure A =
  struct [\rho_A]
  structure B = struct [\rho_B] datatype t [\rho_t] = K end
  functor F(X [\rho_X]; sig type u [\rho_u] val a : u end) =
    struct
      val b : X.u \ast B.t = (X.a, B.K)
    end : sig val b : X.u \ast B.t end
  end
end
```

The relativized tycon, an entity path, can be computed from the static environment at the point of elaboration. For example, when the elaborator reaches the body of the above functor, the static environment is the following:

\([B \mapsto (\rho_B, \{t \mapsto (\rho_t, 0); \emptyset_{ee}\})][X \mapsto (\rho_X, \{u \mapsto (\rho_u, 0), a \mapsto [\rho_u]; \emptyset_{ee}\})]\)

The tycon for \(X.u\) can be relativized by building an entity path, \(\rho_X \rho_u\), by traversing the static environment.

When elaborating the specified type of \(a\) in the functor parameter signature, the reference to the formal tycon \(u\) is relativized to an entity path \(\rho_u\).

After instantiation of the parameter signature during functor elaboration (sec. 7.10), the elaborator must derive the canonical entity path for any newly generated tycons, structures, and functors, which will correspond to an entity variable in the parameter signature. The elaborator uses the canonical entity path when elaborating the functor result signature and functor body.

This after-the-fact entity to entity path mapping is also needed for structure bindings to create the mappings for coerced entities produced during signature matching. Unlike in the functor case, mappings for the entities in the structure binding and the structure entity...
itself must be in scope for the rest of the program.

A policy of fully reducing a tycon before relativizing simplifies the process of relativization. Each primary volatile tycon has a unique declaration site and thus a unique entry in the entity environment from which the elaborator can derive the unique entity path leading to that tycon.

Υ is a sequence of bindings of entity variables to entities. A binding \([ρ \mapsto v]\) in Υ precedes another binding \([ρ' \mapsto v']\) if it comes earlier in the sequence. Note that since entity environments are hierarchical, one binding can precede another even if they are not in the same level in the hierarchy. Let \(\sqsubseteq_{ee}, Υ\) be the total ordering of the entity paths in Υ. Let \(\tilde{ρ}_1 \sqsubseteq_{ee} \tilde{ρ}_2\) only if the entity environment binding denoted by \(\tilde{ρ}_1\) precedes the binding for \(\tilde{ρ}_2\).

Let \(\text{inv}(Υ, τ^n)\) denote the least entity path \(\tilde{ρ}\) with respect to \(\sqsubseteq_{ee}\) such that \(Υ(\tilde{ρ}) = τ^n\).

The definition of relativization uses an operation of the entity environment \(\text{inv}\) to calculate the entity path. Although Υ technically does not admit an inverse because multiple entity paths may map to the same atomic tycon, this operation is intuitively similar to an inverse of Υ lookup.

**Definition 1**

\(\text{inv}(Υ, τ^n)\) is defined to be the entity path associated with the first occurrence (i.e., least binding according to \(\sqsubseteq_{ee}\)) of \(τ^n\) when treating the entity environment Υ as a sequence of bindings.

**Example 1**

Let \(Υ = [ρ_0 \mapsto [ρ_1 \mapsto τ^0_1][ρ_2 \mapsto τ^0_1]].\) \(\text{inv}(Υ, τ^0_1) = ρ_0ρ_1.\)

### 6.4.3 Semantics

Fig. 6.10 formally defines the relativization of syntactic monotypes, tycons, and type expressions. The relativization judgments require an entity environment Υ to interpret atomic tycons associated with those symbolic paths.
$\Gamma \vdash \mathcal{C} \setminus ^{\text{mt}} \mathcal{C}$

$\Gamma \vdash \alpha \setminus ^{\text{mt}} \alpha$  \hspace{1cm} (6.24)

$\rho = \text{inv}(\Gamma, \tau^n) \quad \Rightarrow \quad \Gamma \vdash \check{\mathcal{C}} \setminus ^{\text{mt}} \check{\mathcal{C}}$

$\Gamma \vdash \tau^n(\check{\mathcal{C}}) \setminus ^{\text{mt}} \check{\rho}(\check{\mathcal{C}})$  \hspace{1cm} (6.25)

$\lambda \alpha. \check{\mathcal{C}}(\check{\mathcal{C}}) \uparrow_{\text{mt}} \mathcal{C}_3 \quad \Rightarrow \quad \Gamma \vdash \mathcal{C}_3 \setminus ^{\text{mt}} \mathcal{C}$

$\Gamma \vdash \lambda \alpha. \check{\mathcal{C}}(\check{\mathcal{C}}) \setminus ^{\text{mt}} \mathcal{C}$  \hspace{1cm} (6.26)

$\Gamma \vdash \mathcal{C} \setminus ^{\text{tyc}} \mathcal{C}$

$\Gamma \vdash \mathcal{C} \setminus ^{\text{mt}} \mathcal{C}$

$\Gamma \vdash \lambda \alpha. \check{\mathcal{C}}(\check{\mathcal{C}}) \setminus ^{\text{tyc}} \lambda \alpha. \mathcal{C}$  \hspace{1cm} (6.27)

$\Gamma \vdash \mathcal{T} \setminus ^{\text{te}} \mathcal{T}$

$\Gamma \vdash \mathcal{T} \setminus ^{\text{mt}} \mathcal{T}$

$\Gamma \vdash \mathcal{T} \setminus ^{\text{tyc}} \mathcal{T}$

$\Gamma \vdash \mathcal{T} \setminus ^{\text{te}} \mathcal{T}$  \hspace{1cm} (6.28)

$\Gamma \vdash \forall \alpha. \check{\mathcal{C}} \setminus ^{\text{te}} \forall \alpha. \check{\mathcal{C}}$  \hspace{1cm} (6.29)

**Figure 6.10: Relativization**
Relativization is used in signature expression elaboration of where type (7.7), type definition spec elaboration (rule 7.12), val spec elaboration (rule 7.13), and signature extraction (rules 7.38, 7.39).

**Definition 2 (Well-formedness of entity environments)**

1. If $\Gamma, \emptyset \vdash \bar{\rho} :: \Omega^n \rightarrow \Omega$, then $\Upsilon(\bar{\rho}) = \tau^n$ or $\Upsilon(\bar{\rho}) = \mathcal{C}^\lambda$ such that $|C^\lambda| = n$.

2. If $\bar{\rho}$ is a functor in $\Gamma$, then $\Upsilon(\bar{\rho}) = \psi$.

3. If $\bar{\rho}$ is a structure in $\Gamma$, then $\Upsilon(\bar{\rho}) = R$. 
CHAPTER 7
ELABORATION SEMANTICS

In a first-order module system, functor application can propagate types in the parameter to the functor result. True higher-order semantics is useful for compilation of efficient module code [68]. More importantly, type propagation is necessary to make some sound programs typecheck. THO semantics also reflects the dependent type structure of functors. Types in the functor parameter may also be generative, as in the case of datatypes and types in opaquely ascribed substructures. These generative types have to be faithfully propagated during functor application. Functors complicate type propagation because types in the functor result may be computed from multiple sources. As in the first-order case, types may be locally defined in the result or propagated from the parameter. Applications of formal functors (i.e., functors in the functor parameter) in the functor result should also propagates types. Collectively, generation of fresh instances of generative types and formal functor applications are called functor actions, which must be performed during elaboration of functor application to maximize type propagation.

As in other accounts of module system designs, I present the THO module system as an elaboration semantics. The syntactic module language is elaborated to a semantic representation. Unlike prior accounts, this semantics will use an entity calculus that captures the functor actions in a program. The entity calculus is a third level of representation distinct from the syntactic and semantic representations. It assumes that each type is given a unique name, an entity variable. Unlike other internal languages, the entity calculus only plays a role in elaboration. Entity calculus terms are translated to IL types by the time code reaches the optimization stages of a compiler.

Elaboration accomplishes the following:

1. Produces a static environment mapping variables to static descriptions of values, types, structures, functors, and signatures
2. Produces a typed abstract syntax

3. Produces entity expressions used in representing functor actions

4. Typechecks the program

For simplicity, the discussion of (2) will be postponed to chapter 8. Fig. 7.1 shows the major components of the elaboration process and how they are related.
7.1 Elaboration Representations

Elaboration is mainly concerned with the construction of a static environment. Secondarily, the elaborator produces typed abstract syntax, an entity environment, and an entity expression. The static environment contains the visible static information in the elaborated program. The static information comes in the form of semantic representations of signatures and type information.

The entity environment contains static information that may have been occulted by shadowing and the functor actions describing the production of new static information during functor application. Much of elaboration is concerned with constructing the correct entity environment and using them to interpret entity paths.

7.2 Semantic Objects

A semantic signature is a sequence of signature elements, symbol to spec bindings. Signature elements can be volatile \((s^p)\), definitional \((C^\lambda)\), or a value spec \((T)\). Note that all signature spec elements must be relativized. Nonvolatile elements including definitional tycons \(C^\lambda\)
and value specs $T$ need only specify their fixed static description. Volatile primary specs ($s^p$) such open ones $arity$, structures $\Sigma$, and functors $\Sigma^f$ may be instantiated by signature matching and thus must have a corresponding binding in the entity environment indexed by entity variables. A full signature $M$ gives a full semantic description of a structure. It is comprised of a semantic signature $\Sigma$ and a structure entity $R$ that interprets all the open specifications in $\Sigma$.

A semantic functor signature $\Sigma^f$ binds an entity variable for the functor parameter $\rho$ in the functor result signature. A full functor signature $F$ is comprised a semantic functor signature $\Sigma^f$ and a functor entity $\psi$ that when evaluated will interpret all the open specifications in the functor result signature.

$\gamma$ is a static binding, to which the static environment $\Gamma$ maps identifiers. For value identifiers, the static description is a semantic type expression $T$. For tycon definitions, the static description is $C^\lambda$. There is no static binding for open tycon because all tycons are either defined or instantiated in the static environment. For signature and functor signature identifiers, the static descriptions are semantic signatures ($\Sigma$) and semantic functor signature ($\Sigma^f$). For structure and functor identifiers, the static descriptions are full signatures and full functor signatures respectively. Some static binding (namely for structures and functors) consist of the full signature or full functor signature coupled with the entity variable for that structure or functor. The entity variable is used to construct entity paths during signature extraction and elaboration.

Tycons in signatures and in static environments differ in that tycons in the former will have been relativized during signature elaboration or extraction. Unlike static environments that may be extended throughout the elaboration process, signatures do not change. Hence to ensure that volatile tycons in signatures have an appropriate interpretation, they must be relativized with respect to the entity environment.

In Fig. 7.3, the static environment depicted exhibits the heterogeneous form of the def-
inition. The first layer contains semantic tycons, those types directly bound by the static environment. The second layer consist of the tycons $s$, $u$, and $w$ bound in semantic signatures $\Sigma_0$ and $\Sigma_1$ embedded inside of the static environment. These semantic signatures contain relativized tycons. The full signature for structures also include the structure entity expressions $R_0$ and $R_1$ which define the instantiation of the open tycons in the corresponding semantic signature.

### 7.3 Notation

If $p$ is a symbolic path and $\Sigma$ is a semantic signature, then $p \in \Sigma$ means that following $p$ in $\Sigma$ reaches a spec. If $p$ is a singleton, then $\Sigma$ must contain a binding $p \mapsto \text{spec}$. Otherwise, if $p$ is nonsingleton, then $p = xp'$ such that $x$ is a symbol and $p'$ is a symbolic path such that $\Sigma$ contains a binding $x \mapsto (\rho, \Sigma')$ and $p' \in \Sigma$.

$\Sigma(p)$ is the spec reached by following $p$ in $\Sigma$. The notation $\text{EP}(\Sigma, p)$ denotes the entity path associated with the signature spec referred to by symbolic path $p$. The entity path is
comprised of the entity variables on the path to the element. For example, if the signature is 
\[ \Sigma = [A \mapsto (\rho_A, [B \mapsto (\rho_B, [t \mapsto (\rho_t, 0)]))]) \], then \( \Sigma_{ep}(A.B.t) = \rho_A \rho_B \rho_t \). The same notation is extended to static environments.

Static environment lookup is expressed as \( \Gamma(p) \) where \( p \) is the symbolic path to the static binding. If \( p \) is a singleton, then the static environment lookup would simply return the corresponding static binding. As will be demonstrated below, static environment lookup sometimes requires an entity environment lookup. This is the case when the component being looked up is a definitional tycon in a nonsingleton path. Depending on what kind of entity \( p \) refers to, the lookup is handled differently:

**tycon singleton path** \( p = x \) The semantic tycon is \( \Gamma(x) \).

**nonsingleton** \( p = q.x \) \( \Gamma(q) \) produces a full signature \( (\Sigma_q, R_q) \). Let \( \Upsilon_q = \Upsilon_{clo} \Upsilon_{lcl} \) where \( R_q = \langle \Upsilon_{lcl}, \Upsilon_{clo} \rangle \).

If \( \Sigma_q(x) = (\rho_x, n) \), then \( \Gamma(p) = \Upsilon_q(\rho_x) \).

If \( \Sigma_q(x) = C^\lambda \), then \( \Gamma(p) = \) the interpretation of \( C^\lambda \) under \( \Upsilon_q \).

Note that all tycons are interpreted using the structure entity closest to the occurrence.

**full signature or a full functor signature** \( \Gamma(p) \) denotes a pair \((\bar{\rho}, M)\) such that \( \bar{\rho} \) is the entity path to the structure static binding and \( M \) is the full signature of that structure. For example, for \( \Gamma = [A \mapsto (\rho_A, ([B \mapsto (\rho_B, \Sigma_B)], R_A))] \), \( \Gamma(A.B) = (\rho_A \rho_B, (\Sigma_B, R_A(\rho_A \rho_B))) \).

### 7.4 Elaboration Modes

During elaboration, entity expressions are both produced and consumed to aid in type-checking. Elaboration occurs in two simultaneous modes: the **direct mode** and the **entity compilation mode**. Direct mode consists of core language type checking, direct construction
of entity expressions (such as tycon declaration elaboration and signature instantiation), and evaluation of entity expressions from the static information uncovered during elaboration. The main results of this mode are a static environment mapping symbolic names to static descriptions and an entity environment mapping entity variables to entities. The evaluation of entity expressions may produce new entities which added to static and entity environments. The entity compilation mode compiles functor actions into entity expressions.

Elaboration is entirely a compile-time process. The entity language is distinct from core and module languages. The language is parallel to the semantic representation and the syntactic language. Elaboration translates core and module languages to typed abstract syntax and entity expressions. The part of elaboration that produces entity expressions and declaration is called static entity compilation. Because the entity language is intended to describe functor actions, which are not encoded in the typed abstract syntax, the two are complementary. When the elaborator reaches a functor application in the source, the corresponding application entity expression must be evaluated to produce the static information for the result. Entity expression evaluation yields structure entities.

7.5 Entity Compilation Mode

Besides the direct elaboration mode for the evaluation of entity expressions, described in chapter 6, the elaborator must have an entity compilation mode that produces the entity expressions and semantic representations of the modules from the syntax. The process of the compilation mode is the bulk of the elaboration semantics. Entity expressions for both structures and functors are compiled from the raw implicitly typed abstract syntax trees and the contextual environments used during elaboration. The compilation mode elaboration is the subject of the remainder of this chapter.

The compilation mode elaboration compiles abstract syntax trees to entity expressions. All of elaboration takes place under a static environment $\Gamma$. Recall that semantic signatures
pair all static primary components with entity variables and replace all occurrences of such primaries with a canonical entity path to the corresponding static entity in the entity environment, which is calculated by relativization. Values are classified by their type. Structures are described by their full signature consisting of a structure entity and a signature. A functor static description is a full functor signature, comprised of a functor entity expression (i.e., a $\lambda$-expression) and a functor signature. The main elaboration judgments are the following:

$$\Gamma, \Upsilon, \Sigma \vdash C^s \Rightarrow_{mt} C^s$$  monotype elaboration

$$\Gamma, \Upsilon, \Sigma \vdash C^\lambda \Rightarrow_{tyc} C^\lambda$$  tycon elaboration

$$\Gamma, \Upsilon, \Sigma \vdash T \Rightarrow_{te} T$$  type elaboration

The above judgments produce semantic monotypes/tycons/type expressions from syntactic ones. Tycon elaboration’s sole role is during elaboration of tycon declarations (rule 7.27).

$$\Upsilon \vdash C^s \downarrow_{mt} C^s$$  monotype relativization

$$\Upsilon, \Sigma \vdash C^\lambda \downarrow_{tyc} C^\lambda$$  tycon relativization

$$\Upsilon, \Sigma \vdash T \downarrow_{te} T$$  type expression relativization

The relativization judgment produce relativized tycons from syntactic ones. The entity environment $\Upsilon$ is used to relativize the nonlocally defined tycons. The signature $\Sigma$ is used to construct relativized entity paths for locally defined tycons. Signature extraction and elaboration rely on relativization.
\[\Gamma, \Upsilon, \Sigma \vdash \text{sigexp} \Rightarrow_{\text{sig}} \Sigma'\]  
\text{signature elaboration}  
\[\Gamma, \Upsilon, \Sigma \vdash \text{fsgexp} \Rightarrow_{\text{fsg}} \Sigma^f\]  
\text{functor signature elaboration}  
\[\Upsilon^{\text{clo}}, \Upsilon^{\text{lcl}} \vdash \Sigma \uparrow \Upsilon^{\text{lcl}}\]  
\text{signature instantiation}  
\[\Gamma, \Upsilon \vdash \text{d}^m \Rightarrow_{\text{decl}} (\eta, \Gamma', \Upsilon')\]  
\text{module declaration elaboration}  
\[\Gamma, \Upsilon \vdash \text{strexp} \Rightarrow_{\text{str}} (M, \varphi)\]  
\text{structure expression elaboration}  
\[\Upsilon \vdash \Gamma \hookrightarrow \Sigma\]  
\text{signature extraction}  
\[\Upsilon \vdash (M, \varphi) : \Sigma \Rightarrow_{\text{match}} (M_c, \varphi_c)\]  
\text{signature matching}  
\[\Upsilon \vdash F \preceq \Sigma^f \Rightarrow_{\text{fsgmtch}} (\psi_c, \theta_c)\]  
\text{functor signature matching}  

### 7.6 Type Constructors

The syntactic type language is a subset of the semantic tycon language. Before proceeding, the elaborator must first elaborate the syntactic types to semantic tycons. The elaboration is simply a syntax-directed recursive translation.

The syntactic type language elaborates to the semantic type language. Elaboration interprets the symbolic paths in tycon application \(p(C^s)\) and evaluates the application by standard \(\beta\)-reduction. Note that tycon elaboration \(\Rightarrow_{\text{tyc}}\) elaborates and therefore reduces under \(\lambda\)-abstraction. This ensures that tycon definitions are fully reduced. Because \(\lambda\)s cannot be nested, this reduction is well-defined and normalizing. The normal form is defined as \(C^{n^f}\) in fig. 4.3.

Note that the tycon may be occurring inside a signature spec, in which case it may refer to entity paths for preceding specs.

```plaintext
\text{sig}
\text{type t}
\text{type u = t}
\text{end}
```

During the elaboration of the above, \(t\) will be given an entity variable. When elaborating the occurrence of tycon \(t\) in the definition of \(u\), \(t\) is not the static environment. It can only be found in the partial semantic signature produced by elaborating the open spec for \(t\).
\[ \text{entpath}(\Gamma, \Sigma, p) = \begin{cases} \text{EP}(\Sigma, p) & \text{if } p \in \Sigma \\ \Gamma_{ep}(p) & \text{o.w.} \end{cases} \]
Signature specs may contain relativized tycons, entity paths, referencing entities declared earlier in the same semantic signature. These preceding signature specs (forming a semantic signature \( \Sigma \) itself) help construct an entity path for symbolic paths not defined in the static environment. Rule 7.2 uses \texttt{entpath} to calculate the entity path using static and entity environments. In order to fully relativize such a symbolic path, the \texttt{entpath} metafunction needs to first lookup the symbolic path in the preceding signature specs \( \Sigma \), which contains all the preceding specs in the current signature and only if that fails does it look up in the static environment.

The type elaboration judgments preserve the well-kinding defined in chapter 4.

**Lemma 6 (Monotype Elaboration Preserves Kinding)**

If \( \Gamma, \emptyset \vdash C^s : \Omega \) and \( \Gamma, \Upsilon \vdash C^s \Rightarrow_{mt} \mathcal{E}^s \), then \( \emptyset \vdash_{knds} \mathcal{E}^s : \Omega \).

**Lemma 7 (Tycon Elaboration Preserves Kinding)**

If \( \Gamma, \emptyset \vdash C^{\lambda} : \Omega^n \Rightarrow \Omega \) and \( \Gamma, \Upsilon \vdash C^{\lambda} \Rightarrow_{tyc} \mathcal{E}^{\lambda} \), then \( \emptyset \vdash_{knds} \mathcal{E}^{\lambda} : \Omega^n \Rightarrow \Omega \).

### 7.7 Signature Elaboration

The role of signature elaboration is to relativize tycons, to reduce where type clauses to type definitions, and to decorate all specs corresponding to static entities with entity variables. The elaboration judgment uses static and entity environment to interpret symbolic paths in tycon definitions. However, not all symbolic paths are bound in the static environment. For example, in the following signature, the type definition for \( u \) mentions tycon \( t \) which does not have a corresponding structure entity at this point, hence tycon \( t \) will not be in static and entity environment.

\[
\text{sig type } t \\
\text{type } u = t \\
\text{end}
\]

Rule 7.10 composes the result of elaborating the individual specs. Elaborating subsequent specs in the same signature will require a context (a signature) that includes all the previously elaborated specs \( \Sigma \) and the newly elaborated spec \( \Sigma' \).
\[ \Gamma, \Sigma \vdash \text{sigexp} \Rightarrow \text{sig} \ \Sigma' \]

\[ \Gamma, \Sigma \vdash x \Rightarrow \text{sig} \ (x) \quad (7.6) \]

\[ \Gamma, \Sigma \vdash \text{sigexp} \Rightarrow \text{sig} \ \Sigma' \quad \Sigma'(p) = (\rho, n) \]
\[ \Gamma \vdash C^\lambda \Rightarrow \text{tyc} \ C^\lambda \quad \Gamma, \Sigma \vdash C^\lambda \ \Sigma' \vdash C^\lambda \ \subseteq \text{tyc} \ C^\lambda \]

\[ \Gamma, \Sigma \vdash \text{sigexp} \Rightarrow \text{sig} \ \Sigma' \quad |C^\lambda| = n \]

\[ \Gamma, \Sigma \vdash \text{sigexp} \text{ where type } p = C^\lambda \Rightarrow \text{sig} \ \text{rebind}(p, C^\lambda, \Sigma) \]

\[ \Gamma, \Sigma \vdash \text{specs} \Rightarrow \text{specs} \ \Sigma' \]

\[ \Gamma, \Sigma \vdash \text{sig} \ \text{specs} \Rightarrow \text{sig} \ \Sigma' \]

\[ \Gamma, \Sigma \vdash \text{type} \ t = C^\lambda \Rightarrow \text{spec} \ [t \mapsto C^\lambda] \quad (7.11) \]

\[ \Gamma, \Sigma \vdash \text{val} \ x : T \Rightarrow \text{spec} \ [x \mapsto T] \quad (7.13) \]

\[ \Gamma, \Sigma \vdash \text{sigexp} \Rightarrow \text{sig} \ \Sigma' \quad (\rho \text{ fresh in } \Gamma \text{ and } \Sigma) \]

\[ \Gamma, \Sigma \vdash \text{structure} \ x : \text{sigexp} \Rightarrow \text{spec} \ [x \mapsto (\rho, \Sigma')] \quad (7.14) \]

\[ \Gamma, \Sigma \vdash \text{functor} \ f(X : \text{sigexp}_1) : \text{sigexp}_2 \Rightarrow \text{spec} \ [f \mapsto (\rho, \Pi \rho_x : \Sigma_1, \Sigma_2)] \quad (7.15) \]
Signature spec elaboration produces entity variables for each static component (i.e., open type specs, structure spec, and functor spec) and puts them in the resultant semantic signature. Rules 7.12 and 7.13 relativize semantic tycons and type expressions respectively.

In particular, the signature \( \Sigma \) is needed when relativizing specs in a signature \( \Sigma' \) that contain symbolic paths defined within the same \( \Sigma \). Latter specs can contain symbolic paths declared in earlier specs within the same signature. Because the signature is not yet fully elaborated and does not correspond to any full signature, it is neither defined in the static environment nor the entity environment, each of which only mapping paths defined outside of the signature currently being elaborated.

Rule 7.15 elaborate the formal parameter signature \( \text{sigexp}_1 \) and then the functor body signature \( \text{sigexp}_2 \) with the signature context extended with a spec corresponding to the formal parameter.

Let \( AT(\cdot) \) denote the set of all atomic tycons \( (\tau^n) \) in \( \cdot \). The following lemma guarantees that all tycons are fully relativized by the relativization judgment.

**Lemma 8**

If \( AT(\Sigma) = \emptyset \), for all \( \Sigma_i \in \Gamma.AT(\Sigma_i) = \emptyset \), and \( \Gamma, \Upsilon, \Sigma \vdash \text{sigexp} \Rightarrow_{\text{sig}} \Sigma' \), then \( AT(\Sigma') = \emptyset \).

### 7.8 Signature Instantiation

Signature instantiation produces a free instantiation of a given semantic signature, where free is in the context of both a closure and a local entity environment. Instantiation produces an entity environment that is local, meaning exclusive of closure entity environment bindings (see rule 7.16).

Most of the judgments ignore the signature element. The elements that matter are the those corresponding to static entities. Rule 7.19 produces a fresh atomic tycon \( \tau^n \) of the appropriate arity \( n \). The tycon must be fresh in both closure and local entity environments. Subsequent signature elements must be instantiated in the context of the local entity envi-
\[ \text{Figure 7.7: Signature instantiation} \]

(7.16)

(7.17)

(7.18)

(7.19)

(7.20)

(7.21)
Figure 7.8: Top level declarations

The environment extended with the binding to the new tycon \([\rho \mapsto \tau^n]\). This new binding serves two roles. First, it ensures that \(\tau^n\) is not reused when instantiating the rest of the signature elements. Second, the new binding may be included as part of the closure entity environment when instantiating a structure and functor specs.

Rule 7.20 instantiates a structure spec by recursively instantiating its semantic signature. The structure’s semantic signature may mention either entities in \(\Upsilon^{clo}\) and \(\Upsilon^{lcl}\), that is, preceding entities in the same signature. The resulting entity environment is combined with a closure \(\Upsilon^{clo} \Upsilon^{lcl}\) to form a structure entity. Rule 7.21 instantiates a functor spec by producing a functor entity contains a formal functor functor entity expression \(\lambda \rho x. \Sigma r\) and a closure entity environment.

Lemma 9 (Signature Instantiation Terminates)
\(\Upsilon \vdash \Sigma \vdash \Upsilon'\) terminates.

### 7.9 Top Level Declaration Elaborations

Elaboration of top level declaration is straightforward. The semantics is given in Fig. 7.8. Rule 7.23 elaborates a signature expression and extends the static environment with a binding of the signature name to the semantic signature. Rule 7.24 elaborates module declarations.
using the \( \Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon') \) judgment. The judgment can safely discard the entity declarations from the module declarations because top level declarations will not be wrapped in a functor and therefore lead to functor actions.

### 7.10 Module Elaboration

The module elaboration judgment calls for a static environment and an entity environment as its context. The static environment serves to elaborate tycons and structures, by looking up symbolic paths. The entity environment plays a role in signature elaboration and instantiation of functor parameters.

Fig. 7.9 gives the rules for elaborating module declarations. The judgment computes an entity declaration, a static environment, and an entity environment for the module declarations. The two environments will only contain bindings representing the information in the module declarations and not the closure. Each module declaration elaboration rule calculates a semantic representation of the declaration value (i.e., type expressions, (definitional) tycons, atomic tycons, full signatures, and full functor signatures) and binds that representation in the static environment (and the entity environment if the entity in question is primary).

Rules 7.26 and 7.27 elaborate values to a semantic type expression and syntactic tycon to semantic tycon respectively. Rule 7.26 assumes a core expression elaboration (typechecking) judgment of the form \( \Gamma \vdash e \Rightarrow_{core} s \), which is standard. Neither of these produce any entity declarations and entity environment because they do not contain static entities. Rule 7.28 generates a fresh atomic tycon \( \tau^n \) for a datatype. This atomic tycon is bound in both the static and entity environments. The elaborator further produces an entity declaration \( \rho_t =tyc\ new(n) \) that records the functor action attributed to the datatype declaration. The entity declaration is reserved for evaluation upon functor application of the possible enclosing functor. When evaluated, it will produce another atomic tycon distinct from the \( \tau^n \) generated.
The resultant $\Upsilon$ must be the local entity environment in order for the structure expression judgment for struct $dm$ end to properly construct a structure realization.

Figure 7.9: Module declaration elaboration
\[\begin{align*}
\Gamma, \Upsilon \vdash \text{strexp} & \Rightarrow_{\text{str}} (M, \varphi) \\
\Gamma(p) = (\bar{\rho}, M) \\
\Gamma, \Upsilon \vdash p & \Rightarrow_{\text{str}} (M, \bar{\rho}) \quad (7.31)
\end{align*}\]

\[\begin{align*}
\Gamma, \Upsilon \vdash d^m & \Rightarrow_{\text{decl}} (\eta, \Gamma', \Upsilon^{lcl}) \\
\Upsilon \Upsilon^{lcl} \vdash & \Gamma' \hookrightarrow \Sigma \\
\Gamma, \Upsilon \vdash \text{struct } d^m \text{ end} & \Rightarrow_{\text{str}} ((\Sigma, (\Upsilon^{lcl}, \Upsilon)), (\emptyset)) \quad (7.32)
\end{align*}\]

\[\begin{align*}
\Gamma(p) = (\bar{\rho}, (\Pi X : \Sigma_{\text{par}}, \Sigma_{\text{body}}, \langle \theta; \Upsilon' \rangle)) \\
\Gamma, \Upsilon \vdash \text{strexp} & \Rightarrow_{\text{str}} (M, \varphi) \\
\Upsilon \vdash (M, \varphi) & : \Sigma_{\text{par}} \Rightarrow_{\text{match}} (R_c, \varphi_c) \\
\varphi_{\text{app}} = \bar{\rho}(\varphi_c) & \Upsilon \vdash \varphi_{\text{app}} : \text{str } R_{\text{app}} \\
\Gamma, \Upsilon \vdash p(\text{strexp}) & \Rightarrow_{\text{str}} ((\Sigma_{\text{body}}, R_{\text{app}}), \varphi_{\text{app}}) \quad (7.33)
\end{align*}\]

\[\begin{align*}
\Gamma, \Upsilon \vdash d^m & \Rightarrow_{\text{decl}} (\eta_{\text{def}}, \Gamma_{\text{def}}, \Upsilon_{\text{def}}) \\
\Gamma_{\text{def}}, \Upsilon_{\text{def}} \vdash \text{strexp} & \Rightarrow_{\text{str}} (M, \varphi) \\
\Gamma, \Upsilon \vdash \text{let } d^m \text{ in } \text{strexp} & \Rightarrow_{\text{str}} (M, \text{let } \eta_{\text{def}} \text{ in } \varphi) \quad (7.34)
\end{align*}\]

\[\begin{align*}
\Gamma, \Upsilon, \emptyset_{\text{sig}} \vdash \text{sigexp} & \Rightarrow_{\text{sig}} \Sigma_{\text{spec}} \\
\Gamma, \Upsilon \vdash \text{strexp} & \Rightarrow_{\text{str}} (M_u, \varphi_u) \\
\Upsilon \vdash (M_u, \varphi_u) : \Sigma_{\text{spec}} & \Rightarrow_{\text{match}} (R_c, \varphi_c) \\
\Gamma, \Upsilon \vdash \text{strexp : sigexp} & \Rightarrow_{\text{str}} ((\Sigma_{\text{spec}}, R_c), \varphi_c) \quad (7.35)
\end{align*}\]

\[\begin{align*}
\Gamma, \Upsilon, \emptyset_{\text{sig}} \vdash \text{sigexp} & \Rightarrow_{\text{sig}} \Sigma_{\text{spec}} \\
\Gamma, \Upsilon \vdash \text{strexp} & \Rightarrow_{\text{str}} (M_u, \varphi_u) \\
\Upsilon \vdash (M_u, \varphi_u) : \Sigma_{\text{spec}} & \Rightarrow_{\text{match}} (R_c, \varphi_c) \\
\Upsilon, \emptyset_{\text{spec}} \vdash \Sigma_{\text{spec}} & \Rightarrow_{\text{spec}} \Upsilon_{\text{spec}} \\
\Gamma, \Upsilon \vdash \text{strexp : sigexp} & \Rightarrow_{\text{str}} ((\Sigma_{\text{spec}}, (\Upsilon_{\text{spec}}, \Upsilon)), \varphi_c) \quad (7.36)
\end{align*}\]

Figure 7.10: Structure expression elaboration
here. Rule 7.29 elaborates the structure expression to a full signature and a structure entity expression. Rule 7.30 elaborates a functor declaration via several steps:

1. elaborate the functor parameter signature $\Sigma_x$

2. instantiate it $\Upsilon_x$

3. form a full signature using the result of the previous two steps $(\Sigma_x, R_x)$ where $R_x = \langle \Upsilon_x, \Upsilon \rangle$.

4. elaborate the functor body using a static environment extended with this full signature and the entity environment extended with the structure entity

5. form a full functor signature, a functor entity, and a functor entity expression using the structure entity expression from above

Fig. 7.10 elaborates a structure expression into a full signature and structure entity expression. Rule 7.31 elaborates a structure path. The structure entity expression is an entity path, calculated by accumulating the entity variables in $\Gamma$ leading up to the element denoted by $p$. Rule 7.32 must extract a semantic signature from the static environment produced by elaborating the base structure declarations. Rule 7.33 elaborates a functor application in several steps:

1. looks up the entity path and full functor signature of the functor denoted by symbolic path $p$

2. elaborates the argument structure

3. coerces the argument structure and structure entity expression using the functor parameter signature by signature matching

4. forms the structure entity expression for the application
5. evaluates the structure entity expression using the current entity environment

Rule 7.34 elaborates structure let expressions in the expected way. Rule 7.35 elaborates transparent ascription by signature matching. Rule 7.36 elaborates opaque ascription by signature matching and then instantiating the spec signature to get fresh types for the open type specs.

An important invariant that must be maintained for module declaration elaboration is that all the atomic tycons in the static environment are also in the entity environment. This property guarantees that all atomic tycons can be relativized by the entity environment.

Lemma 10
If \( AT(\Gamma) \subseteq AT(\Upsilon) \) and \( \Gamma, \Upsilon \vdash_d m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon') \), then \( AT(\Gamma') \subseteq AT(\Upsilon') \).

The relationship between the semantic signature \( \Sigma \) and structure entity \( R \) produced by elaboration is precisely related. \( R \) is said to interpret \( \Sigma \), which is defined as follows:

Definition 3
\( \Upsilon \) interprets \( \Sigma \) if for all specs in \( \Sigma \), one of the following must be true:

1. If the spec is open \((\rho, n)\), then \( \Upsilon(\rho) = \tau^n \) or \( \Upsilon(\rho) = C_\lambda \) such that \( \Upsilon \vdash C_\lambda :: \Omega^n \rightarrow \Omega \).

2. If the spec is a structure \((\rho, \Sigma')\), then \( \Upsilon(\rho) = R' \) and \( R' \) interprets \( \Sigma' \).

3. If the spec is a functor \((\rho, \Sigma_f)\), then \( \Upsilon(\rho) = \psi \).

Lemma 11
If \( \Upsilon \) interprets the extracted signature of \( \Gamma \) and \( \Gamma, \Upsilon \vdash strexp \Rightarrow_{str} ((\Sigma, R), \varphi) \), then \( R \) interprets \( \Sigma \).

### 7.11 Signature Extraction

Fig. 7.11 gives the rules for signature extraction, the process of synthesizing a semantic signature for a structure given only the static environment produced by elaborating that
\[ \Upsilon \vdash \emptyset_{sa} \hookrightarrow \emptyset_{sig} \] (7.37)

\[ \Upsilon \vdash \Xi \setminus t^e T \quad \Upsilon \vdash \Gamma \hookrightarrow \Sigma_r \]
\[ \Upsilon \vdash [x \mapsto T] \Gamma \hookrightarrow [x \mapsto T] \Sigma_r \] (7.38)

\[ \Upsilon \vdash \mathcal{E}^\lambda \setminus t^e \mathcal{C}^\lambda \quad \Upsilon \vdash \Gamma \hookrightarrow \Sigma_r \]
\[ \Upsilon \vdash [t \mapsto \mathcal{E}^\lambda] \Gamma \hookrightarrow [t \mapsto \mathcal{C}^\lambda] \Sigma_r \] (7.39)

\[ \text{inv}(\Upsilon, \tau^n) = \rho \quad \Upsilon \vdash \Gamma \hookrightarrow \Sigma_r \]
\[ \Upsilon \vdash [t \mapsto \tau^n] \Gamma \hookrightarrow [t \mapsto (\rho, n)] \Sigma_r \] (7.40)

\[ \Upsilon \vdash \Gamma_r \hookrightarrow \Sigma_r \]
\[ \Upsilon \vdash [x \mapsto (\rho, (\Sigma_1, R_1))] \Gamma_r \hookrightarrow [x \mapsto (\rho, \Sigma_1)] \Sigma_r \] (7.41)

\[ \Upsilon \vdash \Gamma_r \hookrightarrow \Sigma_r \]
\[ \Upsilon \vdash [f \mapsto (\rho, (\Sigma_f^1, \psi))] \Gamma_r \hookrightarrow [f \mapsto (\rho, \Sigma_f^1)] \Sigma_r \] (7.42)

Figure 7.11: Signature extraction
structure and an entity environment. Because the signature must contain only relativized type expressions and tycons, rules 7.38 and 7.39 must relativize the semantic type expression and tycon in the static environment. Rule 7.40 produces a spec from an atomic tycon binding, which specifies the arity $n$ of the tycon. The rule looks up the atomic tycon in the “inverse” of $\Upsilon$. This rule assumes that $\Upsilon$ and $\Gamma$ are synchronized. In particular, since $t$ is a local name in the static environment (i.e., its binding is not nested inside some signature in $\Gamma$), $\tau^n$ should also be local in $\Upsilon$ and therefore have a singleton entity path.

Lemma 12 (Synchronization of Static and Entity Environments)

If $\Gamma, \Upsilon \vdash d^n \Rightarrow decl(\eta, \Gamma', \Upsilon')$ and $EV(\Gamma) \subseteq dom(\Upsilon)$, then $EV(\Gamma') \subseteq dom(\Upsilon')$.

For full signatures and full functor signatures, signature extraction is simple. Rules 7.41 and 7.42 forms structure and functor specs by projecting out the entity variable and semantic signature (or semantic functor signature) from the static environment mapping for the structure and functor respectively.

7.12 Signature Matching

Signature matching plays three principal roles during elaboration. It coerces functor arguments to the form dictated by the functor parameter signature in rule 7.33. Rules 7.35 and 7.36 use signature matching to constrain the full signature of the structure for transparent and opaque ascription respectively. Figs. 7.12 and 7.13 describe the signature matching judgment and its subsidiary judgment respectively. The main judgment for signature match-
\( \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \)

\[ (7.44) \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash \emptyset_{\text{sig}} \Rightarrow \text{coerce} (\emptyset_{\text{ee}}, \bullet) \]

\[ (7.45) \]

\[ \Upsilon \vdash \Sigma_a(x) \equiv \top \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash [x \mapsto \top] \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ (7.46) \]

\[ \Sigma_a(t) = (\rho_a, n') \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash [t \mapsto (\rho, n)] \Sigma_s \Rightarrow \text{coerce} ([\rho \mapsto \Upsilon(\rho_a)] \Upsilon', [\rho = \text{def} \rho_a \rho_u \eta]) \]

\[ (7.47) \]

\[ \Sigma_a(t) = C^\lambda \]

\[ |C^\lambda| = n \]

\[ \Upsilon \vdash C^\lambda = C^\lambda \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash [t \mapsto C^\lambda] \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ (7.48) \]

\[ \Sigma_a(t) = (\rho, n) \]

\[ n = |C^\lambda_s| \]

\[ \Upsilon \vdash C^\lambda_s = \Upsilon(\rho) \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ \Upsilon, \Sigma_a, \rho_a \vdash [t \mapsto C^\lambda_s] \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ (7.49) \]

\[ \Sigma_a(x) = (\rho_x, \Sigma_x) \]

\[ \Upsilon \vdash (\Sigma_x, \Upsilon(\rho_x), \rho_u \rho_x) : \Sigma_s \Rightarrow \text{match} ((\Sigma_c, R_c), \varphi_c) \]

\[ \Upsilon[\rho_s \mapsto R_c], \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta') \]

\[ (7.50) \]

\[ \Sigma_a(f) = (\rho_a, \Sigma^f_a) \]

\[ \Upsilon \vdash (\Sigma^f_a, \Upsilon(\rho_a), \rho_u \rho_a) : \Sigma^f_s \Rightarrow \text{fsgmtch} (\psi_c, \theta_c) \]

\[ \Upsilon[\rho_s \mapsto \psi_c], \Sigma_a, \rho_a \vdash \Sigma_s \Rightarrow \text{coerce} (\Upsilon', \eta) \]

\[ (7.51) \]

Figure 7.13: Structure signature matching
ing \( \Upsilon \vdash (M, \varphi) : \Sigma \Rightarrow_{match} (R_c, \varphi_c) \) forms the coercion structure entity expression \( \varphi_c \) which binds the uncoerced, original entity expression to a fresh entity variable \( \rho_u \) for use by the coercion entity declarations \( \eta \). In general, \( \eta \) may alias components of \( \varphi \) as is in the case in rule 7.46. The coercion structure entity expression needs both \( \varphi \) and \( \eta \) because the elaborator needs to evaluate \( \varphi \) at least for its functor actions and \( \eta \) are the actual exported entity variables. The entity environment for signature matching is a composite of the context \( \Upsilon \), actual structure local environment \( \Upsilon_a \), and actual structure closure environment \( \Upsilon_{clo} \). All three are necessary because \( \Upsilon \) closes the spec signature \( \Sigma_s \), \( \Upsilon_a \) the actual \( \Sigma_a \), and \( \Upsilon_{clo} \) the local environment \( \Upsilon_a \).

A subsidiary judgment \( \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{coerce} (\Upsilon', \eta) \) matches actual structure components to individual specs. The actual structure signature \( \Sigma_a \) is used to match against each spec. The entity variable \( \rho_u \) provides access to the uncoerced structure entity expression in case the elaborator needs to alias components. The judgment will produce both a semantic signature and a local entity environment, used by the match judgment to produce a coerced full signature.

Rule 7.44 matches against an empty spec signature, in which case the coercion signature, entity environment, and entity declarations are empty. Rule 7.45 matches value specs only when the value’s type expressions are equivalent when interpreted under the entity environment. For open tycon specs, there are two cases. Rule 7.46 matches an open tycon spec against an open tycon spec in actual. In that case, the rule will use the structure entity for the actual and an alias to the entity path for the actual \( \rho_u \rho_a \) for the entity declaration. Rule 7.47 matches an open tycon spec against a definitional tycon spec in the actual. The arities of the definitional tycon and the open tycon spec must match up. Rule 7.48 matches a definitional tycon spec in the spec signature to a definitional tycon spec in the actual. Again, this match requires the equivalence of the spec and actual definitional tycons interpreted under the entity environment. Rule 7.49 matches a definition tycon spec in the
spec signature to an open tycon in the actual, which should be consistent with the tycon in entity environment $\Upsilon(\rho)$. Rules 7.50 and 7.51 match structure and functor specs respectively. Being specs for entities, these rules must produce a new entity environment binding and entity declaration. Rule 7.50 forms the bindings using the full signature and structure entity expression from performing signature matching on the signature in the structure spec. The full signature the rule will match against is formed by the spec signature $\Sigma_x$ and the corresponding structure entity $\Upsilon(\rho_x)$. Rule 7.51 works similarly.

The relationship between the resultant structure entity, the spec signature, and actual signature is precise:

**Lemma 13**

If $\Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{\mathsf{match}} (R_c, \varphi_c)$, then for all $[x \mapsto s] \in \Sigma_s$, $[x \mapsto s'] \in \Sigma_a$ such that $R_c(s) = R_a(s')$.

An important property is that the structure entity expression constructed by signature matching should evaluate to the coerced entity environment up to isomorphism. After all, the role of the structure entity expression is to evaluate to an entity environment that instantiates the spec signature to an actual argument structure in the current entity environment.

**Lemma 14**

If $\Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{\mathsf{match}} (R_c, \varphi_c)$, then $\Upsilon \vdash \varphi_c \Downarrow_{\mathsf{str}} R'$ such that $R_c$ is isomorphic to $R'$.

### 7.13 Functor Signature Matching

Fig. 7.14 defines the functor signature matching judgment $\Rightarrow_{\mathsf{fsgmtch}}$ which is composed of a two rules, rule 7.52 and 7.53. Both rules calculate a coerced functor entity $\psi_c$ and functor entity expression $\theta_c$. To match the functor spec, the functor spec’s parameter signature $\Sigma_{spar}$ is matched against a full signature formed by the argument signature $\Sigma_{apar}$ and the instantiation of $\Sigma_{apar}$ plus a structure entity expression for the argument. Rule 7.52 uses the
structure entity expression \( \varphi \) for the a standard functor entity expression \( \lambda \rho.\varphi \). Rule 7.53 only has a formal functor entity expression \( \lambda \rho.\Sigma \). The functor spec’s result signature \( \Sigma_{sres} \) is then instantiated and that instantiation is used to form a full signature for the functor result in order to match against the actual functor spec’s result signature \( \Sigma_{ares} \). The resultant structure entity expression can be used to form the final coerced \( \psi_c \) and \( \theta_c \).
CHAPTER 8
TRANSLATION

One way to show the soundness of the module system is to provide a translation to an applied $F_\omega$ language. What this translation amounts to is a decoupling of the static content from the dynamic content of modules. Module systems that permit decoupling are said to admit phase separation.

The elaboration semantics given in the previous chapter is sufficient and self-contained. However, to support translation into $F_\omega$, the elaboration must be instrumented to annotate module language terms with their full signatures, full functor signatures, signature matching coercions, and free instantiation of functor parameters.

8.1 Annotated Module Language

Fig. 8.1 gives the grammar for the module language annotated with semantic objects from elaboration. The semantic objects are enclosed by angle brackets $\langle\rangle$. Structure paths must be annotated with the entity path corresponding to that symbolic path. The functor path in a functor application should be annotated with its entity path and the entity variable for the formal parameter. The entity variable for the formal parameter is necessary for producing the necessary coercions to get the type and value content of an argument structure into the proper form. The argument structure should be annotated with its coercion full signature and coercion structure entity expression. The associated entity variable should annotate both structure and functor declarations. The functor declaration further requires the annotation of a full functor signature and free instantiation entity environment. All this information is readily available during the elaboration process. Thus, annotating is straightforward. Below, I show a select modified rule:
Core declarations

\[ \begin{align*} 
  d^c & ::= \text{val } x = e \mid \text{type } t = C^\lambda \mid \text{datatype } \hat{\alpha} t \\
  \text{spec} & ::= \text{structure } x : \text{sigexp} \mid \text{type } \vec{\alpha} t \\
  & \quad \mid \text{functor } f(x : \text{sigexp}_1) : \text{sigexp}_2 \\
  & \quad \mid \text{val } x : T \\
  \text{specs} & ::= \emptyset \text{specs} \mid \text{spec, specs} \\
  \text{sigexp} & ::= x \mid \text{sig specs end} \mid \text{sigexp where type } p = C^\lambda \\
  \text{strexp} & ::= \langle\langle \vec{\rho} \rangle\rangle \mid \text{struct } d^m \mid \langle\langle \vec{\rho}, \rho_{\text{par}} \rangle\rangle (\text{strexp}(\langle\langle M, \rho_u \rangle\rangle)) \\
  & \quad \mid \text{strexp : sigexp} \mid \text{strexp := sigexp} \\
 \end{align*} \]

Module decls

\[ \begin{align*} 
  d^m & ::= \circ \mid \text{structure } x(\langle\rho\rangle) = \text{strexp}, d^m \\
  & \quad \mid \text{functor}(\langle\rho, F, \Upsilon\rangle) f(x : \text{sigexp}) = \text{strexp}, d^m \mid d^c, d^m \\
\end{align*} \]

Top level decls

\[ \begin{align*} 
  d^t & ::= \circ \mid \text{signature } x = \text{sigexp}, d^t \mid d^m, d^t \\
\end{align*} \]

Figure 8.1: Annotated module language

\[ \Gamma(p) = (\vec{\rho}, (\Pi \rho_{\text{par}} : \Sigma_{\text{par}}. \Sigma_{\text{body}}, (\theta; \Upsilon')) ) \quad \Gamma, \Upsilon \vdash \text{strexp} \Rightarrow_{\text{str}} (M, \varphi) \]

\[ \Upsilon \vdash (M, \varphi) : \Sigma_{\text{par}} \Rightarrow_{\text{match}} (M_c, \text{let } \rho_u = \varphi \text{ in } \langle\eta\rangle) \]

\[ \varphi_{\text{app}} = \vec{\rho}(\varphi_c) \quad \Upsilon \vdash \varphi_{\text{app}} \downarrow_{\text{str}} R_{\text{app}} \]

\[ \Gamma, \Upsilon \vdash p(\text{strexp}) \Rightarrow_{\text{str}} (p(\langle\vec{\rho}, \rho_{\text{par}}\rangle)(\text{strexp}(\langle\langle M_c, \rho_u \rangle\rangle)), (\Sigma_{\text{body}}, R_{\text{app}}, \varphi_{\text{app}})) \quad (8.1) \]

As shown in the above rule, the entity path for the functor and the functor parameter entity variable are in the static environment at elaboration time. The coercion full signature \(M_c\) and entity variable \(\rho_u\) for the uncoerced argument structure.

8.2 Target Language: \(F_\omega\)

Fig. 8.2 gives the grammar for the applied \(F_\omega\). The language is nearly identical to the one Shao used [68]. It is an \(F_\omega\) core enriched with record types and declarations.

An \(F_\omega\) kind \(k\) is either a monokind or an arrow kind. A tycon \(c\) can be a type variable, an atomic tycon \(\text{inj}(\tau^n)\), an explicitly kinded tycon-level \(\lambda\) abstraction, and a tycon application.
The tycon for a record is represented as \( \{ l_0 : c_0, \ldots, l_n : c_n \} \) where \( l_i \)'s are record field labels and \( c_i \)'s are the tycons for the fields corresponding to those labels. The tycon form \( c.l \) can project out the field type of field \( l \) assuming that \( c \) is a record type containing a field of that label. Type expressions \( t \) are either a tycon or an explicitly quantified and kinded polymorphic type \( \forall \alpha :: k.t \). Unlike the version of the \( F_\omega \) type system given by Shao, this type system includes atomic tycons. Also, this type system simplifies Shao’s by subsuming base types (int) and arrow types \( \tau \rightarrow \tau' \) using an atomic tycon and type application \( \rightarrow^2 (\tau)(\tau') \).

Types \( t \) omit arrow and record types from Shao’s version.

\( F_\omega \) expressions include variables, an explicitly-typed \( \lambda \)-abstractions, applications of expressions, an explicitly-kindred type abstraction \( \Lambda \alpha :: k.e_\omega \), and type application \( e_\omega[c] \). An expression can also be a record expression \( \{ l_0 = e_{\omega 0}, \ldots, l_n = e_{\omega n} \} \), a record field projection \( e_\omega.l \), or a let expression. Within let expressions are declarations, which are sequences of variables bound to expressions terminated by \( \cdot \), the empty declaration. The same environment is used for both kinding \( \alpha :: k \) and typing \( x : t \). Note that the abstraction \( (\lambda x : t.e_\omega) \) and type application \( e_\omega[t] \) both permit application to polymorphic types \( \forall \alpha :: k.t \). This is necessary because functors and functor applications in the module language support abstraction over polymorphic types \( (\text{functor } F(X : \text{sig val id :: } \forall \alpha.\alpha \rightarrow \alpha \text{ end}) = \text{struct end}) \) and application to such types. When these are translated to \( F_\omega \), the polymorphism must be maintained.
Fig. 8.3 describes well-kinding of $F_\omega$ tycons. The kind system is standard. The judgment relies only on the kind part of the environment. Rule 8.3 gives the curried kind for the $n$-ary atomic tycon. The resultant kind is curried $n$ times. Rules 8.7 and 8.8 kind record tycons and record projection tycons respectively in the usual way.

8.2.1 Kind System

Fig. 8.4 gives the type system for $F_\omega$. $\vdash t \text{ type}$ is an axiom. In particular $\vdash k \text{ type}$ is not true. Type application (rule 8.13) assumes that the left-hand side has a type $\lambda\alpha.k.t'$. The type of the type application is $t'{t/\alpha}$. Rule 8.14 gives the record tycon $\{l_0 : c_0, \ldots, l_n : c_n\}$
\[
\begin{align*}
E &\vdash e_\omega : t \\
\vdash E(x) \text{ type} \\
E &\vdash x : E(x) \\
E &\vdash e_\omega : t' \\
E &\vdash (\lambda x : t.e_\omega) : t \rightarrow t' \\
E &\vdash e_\omega_1 : t_1 \rightarrow t_2 \\
E &\vdash e_\omega_2 : t_1 \\
E &\vdash e_\omega_1(e_\omega_2) : t_2 \\
E &\vdash [\alpha :: k] e_\omega : t \\
E &\vdash (\Lambda \alpha :: k.e_\omega) : (\forall \alpha :: k.t) \\
E &\vdash e_\omega : \forall \alpha :: k.t' \\
E &\vdash c :: k \\
E &\vdash e_\omega[c] : t'[c/\alpha] \\
E &\vdash e_\omega_0 : t_0 \ldots E &\vdash e_\omega_n : t_n \\
E &\vdash \{ l_0 = e_\omega_0, \ldots, l_n = e_\omega_n \} : \{ l_0 : t_0, \ldots, l_n : t_n \} \\
E &\vdash e_\omega : \{ \ldots, l_i : t_i, \ldots \} \\
E &\vdash e_\omega[l_i : t_i] \\
E &\vdash d :: dec \ E' \quad E' &\vdash e_\omega : t \\
E &\vdash \text{let } d \text{ in } e_\omega : t \\
E &\vdash c :: k \quad E &\vdash \alpha :: k e_\omega : t \\
E &\vdash \text{let } tyc \ \alpha = c \ \text{in } e_\omega :: t \\
E &\vdash d :: dec \ E' \\
E &\vdash e_\omega : \emptyset_{\omega dec} :: dec \ E \\
E &\vdash e : t \\
E &\vdash x : t \\
E &\vdash e[x : t] :: dec \ E' \\
E &\vdash x = e :: dec \ E' \\
\end{align*}
\]

Figure 8.4: $F_\omega$ Type System
for a record expression \( \{ l_0 = e_0, \ldots, l_n = e_n \} \) as long as all its field expressions have the correct type, \textit{i.e.}, \( e_i : t_i \).

The notation \( \{ c/\alpha \}_{tyc} \) denotes tycon substitution. This kind of substitution does not act on values that may coincidentally share the same name \( \alpha \). This distinct form of substitution is necessary because tycons and values will have distinct namespaces.

### 8.3 Translation Semantics

The translation rules all assume an annotated module language that was produced by an error-free elaboration. Since the program was successfully elaborated, the semantics can make some assumptions about the program, especially what variables will or will not be accessed, and thereby simplify the rules.

The notation \( \text{dom}(d) \) here denotes the set of all the bound names, \textit{i.e.}, the \( \rho \) in \( \rho = e \).

Fig. 8.5 gives the rules for translating module declarations to F\( \omega \) declarations. Rule 8.20 is the empty declaration case. Rule 8.21 translates the core language expression \( e \) to a corresponding \( e_{\omega} \). Unlike the static entity case, shadowing a previous variable \( x \) is harmless so \( x \) will be used as the variable to bind. Rules 8.22 and 8.23 do not produce F\( \omega \) expressions because type definitions and datatype declarations play no role in the dynamic semantics.

Rules 8.24 and 8.25 must produce declarations binding the entity variable of the structure or functor instead of the syntactic variable. The rule 8.25 will use the judgment \( \vdash \Sigma \sim_{\text{sig}} k \) defined in Fig. 8.7 to extract the kind information from the semantic signature for the functor parameter. The type for the F\( \omega \) \( \lambda \) abstraction is calculated separately from the same semantic signature.

Fig. 8.6 gives the translation of the module language to F\( \omega \). Rule 8.29 translates structure applications. The functor being applied can be translated by translating its entity path to the corresponding F\( \omega \) record projection form. The actual argument structure expression is translated to \( e_{\omega} \), but the functor is expecting a parameter of the form defined in full
\[
\text{decl} ::= \text{val } x = e \mid \text{type } t = C^\lambda \mid \text{datatype } \vec{\alpha} \ x \mid \text{structure } x(\langle \rho \rangle) = \text{strexp} \\
\mid \text{functor } f(\langle \ldots \rangle)(X : \text{sigexp}) = \text{strexp}
\]

\[
\begin{align*}
\circ & \sim_{\text{dec}} \emptyset_{\omega_{\text{dec}}} & (8.20) \\
\text{val } x = e & \sim_{\text{dep}} x = e_\omega & (8.21) \\
\text{type } t & = C^\lambda \sim_{\text{dec}} \emptyset_{\omega_{\text{dec}}} & (8.22) \\
\text{datatype } \vec{\alpha} \ x & \sim_{\text{dec}} \emptyset_{\omega_{\text{dec}}} & (8.23) \\
\text{strexp} & \sim_{\text{exp}} e_\omega & (8.24) \\
\text{structure } x(\langle \rho \rangle) & = \text{strexp} \sim_{\text{dec}} \rho = e_\omega \\
F = (\Pi_{\rho_x : \Sigma_{\text{par}}} \Sigma', \psi) & \vdash \Sigma_{\text{par}} \sim_{\text{knd}} k \quad \Upsilon_{\text{ins}} \vdash \Sigma_{\text{par}} \sim_{\text{type}} t \\
\text{strexp} & \sim_{\text{exp}} e_\omega & (8.25) \\
\text{functor } f(\langle \rho, F, \Upsilon_{\text{ins}} \rangle)(X : \text{sigexp}) & = \text{strexp} \\
\sim_{\text{dec}} \rho & = \Lambda \rho_x :: k.\lambda \rho_x : t.e_\omega \\
\text{decl} & \sim_{\text{dec}} d_1 \quad \text{d}^m & \sim_{\text{dec}} d_2 \\
\text{decl, } \text{d}^m & \sim_{\text{dec}} d_1; d_2 & (8.26)
\end{align*}
\]

Figure 8.5: Translation for declarations
The F\textsubscript{ω} kinds, tycons, and types for tycon content of structures, type content of structures, and structures themselves are record kinds, record tycons, and record types respectively. Each label-field pair is a pair of an entity variable and the kind, tycon, and type of the entity referred to by that entity variable. Let lks, lcs, and lts be a sequence of record label-field for kinds, tycons, and types respectively. The notation lks \uplus lks' appends two record kinds. The notation is extended in the usual way to append tycons and types. No
shadowing is expected because all records field names will be entity variables.

Fig. 8.7 gives the rules for producing $F_\omega$ kinds from semantic signatures. Value specs and type definition specs do not contribute to the kind (rules 8.33 and 8.34). Open tycon specs do contribute. Rule 8.35 produces the kind field $\rho :: \Omega^n \rightarrow \Omega$ such that $n$ is the arity of the open tycon spec. Rules 8.36 and 8.37 construct kinds for structure and functor specs from the semantic signature and functor signature respectively.

Fig. 8.8 defines the judgment that calculates the tycons from the full signature of a structure. Again, value specs, and type definition specs do not contribute to the tycon (rules 8.40 and 8.41). Rule 8.42 translates an open spec by looking up the entity environment for the tycon entity associated with the open tycon spec. The rule 8.43 translates the structure spec signature in the context of the entity environment calculated from the structure entity $(\Upsilon(\rho) = \langle \Upsilon^{lcl}, \Upsilon^{clo} \rangle = \Upsilon^{clo}\Upsilon^{lcl})$. To calculate the tycon for a functor component, rule 8.44 must supply rule 8.45 the functor signature and entity from looking up the entity environ-
\[ \vdash M \sim_{\text{tyc}} c \]

\[ \Upsilon \vdash \Sigma \sim_{\text{spec}} \text{lcs} \]

\[ \Upsilon \vdash \emptyset_{\text{sig}} \sim_{\text{spec}} \{ \} \] \hspace{1cm} (8.39)

\[ \Upsilon \vdash \Sigma \sim_{\text{spec}} \text{lcs} \]

\[ \Upsilon \vdash [x \mapsto \xi] \Sigma \sim_{\text{spec}} \text{lcs} \] \hspace{1cm} (8.40)

\[ \Upsilon \vdash \Sigma \sim_{\text{spec}} \text{lcs} \]

\[ \Upsilon \vdash [x \mapsto \lambda] \Sigma \sim_{\text{spec}} \text{lcs} \] \hspace{1cm} (8.41)

\[ \Upsilon \vdash \Sigma \sim_{\text{spec}} \text{lcs} \]

\[ \Upsilon \vdash [x \mapsto (\rho, n)] \Sigma \sim_{\text{spec}} \rho = \text{inj}(\Upsilon(\rho)), \text{lcs} \] \hspace{1cm} (8.42)

\[ \Upsilon(\rho) \vdash \Sigma \sim_{\text{spec}} \text{lcs} \]

\[ \Upsilon \vdash [x \mapsto (\rho, \Sigma)] \Sigma' \sim_{\text{spec}} \rho = \{\text{lcs}\}, \text{lcs'} \] \hspace{1cm} (8.43)

\[ \vdash (\Sigma^f, \Upsilon(\rho)) \sim_{\text{fctspec}} c \]

\[ \Upsilon \vdash [x \mapsto (\rho, \Sigma^f)] \Sigma \sim_{\text{spec}} \rho = c, \text{lcs'} \] \hspace{1cm} (8.44)

\[ \vdash F \sim_{\text{fctspec}} c \]

\[ \vdash (\Pi \rho: \Sigma, \Sigma'; (\lambda \rho, \varphi; T)) \sim_{\text{fctspec}} \lambda \rho :: k.c \] \hspace{1cm} (8.45)

---

Figure 8.8: F\(_\omega\) tycon synthesis
\[
\Gamma \vdash \text{nf} \cong \text{nf}_{\text{type}} \; t
\]

\[
\Gamma \vdash \alpha \cong \text{nf}_{\text{type}} \; \alpha
\]  
(8.46)

\[
\Gamma \vdash \text{nf} \cong \text{nf}_{\text{type}} \; t_1 \ldots \Gamma \vdash \text{nf} \cong \text{nf}_{\text{type}} \; t_n
\]

\[
\Gamma \vdash \tau^n \left( \text{nf} \right) \cong \text{nf}_{\text{type}} \; \text{inj} (\tau^n) (t_1) \ldots (t_n)
\]  
(8.47)

\[
\Delta, \Upsilon \vdash \text{cs} \cong \text{type} \; t
\]

\[
\alpha \in \Delta
\]

\[
\Delta, \Upsilon \vdash \alpha \cong \text{type} \; \alpha
\]  
(8.48)

\[
\text{cs} \text{ s.t. } \Delta, \Upsilon \vdash \text{cs} \; \parallel_{\text{mt}} \; \text{cs} \; \forall \text{cs} \in \text{cs}
\]

\[
\Upsilon (\hat{\rho}) (\text{cs}) \; \parallel_{\text{mt}} \; \text{nf} \; \vdash \text{nf} \cong \text{nf}_{\text{type}} \; t
\]

\[
\Delta, \Upsilon \vdash \hat{\rho} \left( \text{cs} \right) \cong \text{type} \; t
\]  
(8.49)

\[
\Upsilon \vdash \text{t} \cong \text{type} \; t
\]

\[
\emptyset_{\text{knd}}, \Upsilon \vdash \text{cs} \cong \text{type} \; t
\]

\[
\Upsilon \vdash \text{typ} (\text{cs}) \cong \text{type} \; t
\]  
(8.50)

\[
\left[ \alpha \right], \Upsilon \vdash \text{cs} \cong \text{type} \; t
\]

\[
\Upsilon \vdash \forall \alpha. \text{cs} \cong \text{type} \; \forall \alpha. \text{t}
\]  
(8.51)

Figure 8.9: $F_\omega$ synthesis of types from normal form semantic tycons, relativized monotypes, and type expressions
\[ \therefore \vdash \Sigma \rightsquigarrow_{\text{type}} \text{lts} \]

\[ \therefore \vdash \emptyset_{\text{sig}} \rightsquigarrow_{\text{type}} \{\} \]  
(8.52)

\[ \therefore \vdash \mathcal{T} \rightsquigarrow_{\text{type}} t \quad \therefore \vdash \Sigma' \rightsquigarrow_{\text{type}} \text{lts} \]
\[ \therefore \vdash [x \mapsto \mathcal{T}]\Sigma' \rightsquigarrow_{\text{type}} x : t, \text{lts} \]  
(8.53)

\[ \therefore \vdash \Sigma' \rightsquigarrow_{\text{type}} \text{lts} \]
\[ \therefore \vdash [x \mapsto \mathcal{C}\lambda]\Sigma' \rightsquigarrow_{\text{type}} \text{lts} \]  
(8.54)

\[ \therefore \vdash \Sigma' \rightsquigarrow_{\text{type}} \text{lts} \]
\[ \therefore \vdash [x \mapsto (\rho, n)]\Sigma' \rightsquigarrow_{\text{type}} \text{lts} \]  
(8.55)

\[ \therefore \vdash \Sigma(\rho) \rightsquigarrow_{\text{type}} \text{lts} \quad \therefore \vdash \Sigma' \rightsquigarrow_{\text{type}} \text{lts}' \]
\[ \therefore \vdash [x \mapsto (\rho, \Sigma)]\Sigma' \rightsquigarrow_{\text{type}} \rho : \{\text{lts}\}, \text{lts}' \]  
(8.56)

\[ \therefore \vdash \Sigma_x \rightsquigarrow \text{knd} k \quad \therefore \vdash \Sigma_x \rightsquigarrow_{\text{type}} \text{lts}_x \]
\[ \therefore \vdash \Sigma \rightsquigarrow_{\text{type}} \text{lts} \quad \therefore \vdash \Sigma' \rightsquigarrow_{\text{type}} \text{lts}' \]
\[ \therefore \vdash [x \mapsto (\rho, \Pi \rho_x : \Sigma_x, \Sigma)]\Sigma' \rightsquigarrow_{\text{type}} \rho : (\forall \rho_x : k.\{\text{lts}_x\} \rightarrow \{\text{lts}\}), \text{lts}' \]  
(8.57)

Figure 8.10: F_\omega type synthesis
\[
\text{coerce}(\rho_u, \text{lt} \text{ts}) = \begin{cases} 
\{\} & \text{lt} \text{ts} = \{\} \\
\{l_0 = \rho_u, l_0\} \uplus \text{coerce}(\rho_u, \text{lt} \text{ts}') & \text{lt} \text{ts} = \{l_0 : t_0\} \uplus \text{lt} \text{ts}' 
\end{cases}
\] (8.58)

Figure 8.11: Coercion of dynamic components

Fig. 8.11 defines the function \text{coerce} that coerces an \(F_\omega\) expression \(e_\omega\) to a record type \text{lt} \text{ts}. It thins out the value part by dropping components not in the specified type. The resultant expression should have the target type \{\text{lt} \text{ts}\}\{c/\rho\}\text{ty}c where \text{lt} \text{ts} is parameterized by \(\rho\). The type-level coercion was already taken care of by \(F_\omega\) tycon synthesis.

**Lemma 15 (Correctness of Value-Level Coercion)**

If \(E \vdash \{\text{lt} \text{ts}_u\} : k, E \vdash \rho_u : \{\text{lt} \text{ts}_u\}\), for all \(l \in \text{dom}(\text{lt} \text{ts}_u) \cap \text{dom}(\text{lt} \text{ts}_c)\) \(t_c\{c/\alpha\} = t_u\) such that \(l : t_c \in \text{lt} \text{ts}_c\) and \(l : t_u \in \text{lt} \text{ts}_u\), and \text{coerce}(\rho_u, c, \text{lt} \text{ts}_c) = e_\omega\), then \(E \vdash e_\omega : \{\text{lt} \text{ts}_c\}\{c/\alpha\}\).
8.4 Soundness

Theorem 1

If a closed structure expression \( \text{strexp} \) elaborates to \((M, \_))\) and \( \text{strexp} \) translates to \(e_\omega\), then

\[ \vdash e_\omega : t \text{ and } \vdash M \Rightarrow_{\text{type}} t. \]

The proof of the above lemma is sketched out in more detail in the proof appendix. It relies on much of the same machinery as Shao’s proof \([68]\). The proof diverges in Shao’s memoization of pre-translated \(F_\omega\) tycon and types in the functor representation during elaboration of the functor. Furthermore, this proof must deal with the coercive signature matching semantics which is not present in Shao’s main elaboration-translation semantics.

8.5 Type Generativity

Shao’s calculus and many of the related accounts of the module system do not fully support the type generativity present in ML. Type generativity stems from two constructs, opaque ascription and datatype declarations.

Seemingly benign examples that feature datatype declarations are not supported.

```plaintext
structure M =
struct
    datatype t
    val id : t -> t = fn x:t => x
end

functor F() =
struct
    datatype t
    val id : t -> t = fn x:t => x
end
```

Datatype declarations are an interesting case because unlike the rest of the static content of a module, datatype declarations translate to \(F_\omega\) tycons that are not referentially transparent. Whereas all other forms of \(F_\omega\) tycons can be duplicated and be supplied as arguments to type applications, the tycons from datatype declarations cannot. This is because the identity of the tycon from a datatype must be maintained. The tycon cannot be copied. Consequently, such a tycon must be bound (by the special let tyc form) to a type
variable that is in the scope of all subsequent occurrences. It turns out that in general, the expressions constructing these tycons are exactly the entity expressions from the module system. The referentially transparent static content such as type definitions can continue to be packaged together in a tycon record to which \(\Lambda\) type abstractions can be directly applied.

Another difference between datatype declarations and type definitions is what is translated into \(F_\omega\). For datatype declarations, the entity expression is translated. For type definitions, the full signature is directly translated to an \(F_\omega\) tycon.

Translation decomposes functors into a tycon part and a value part. The hat notation \(\hat{x}\) denotes the variable is a type variable that refers to the type part of a structure or functor \(x\). Let \(k\) represent \(\Omega^0 \rightarrow \Omega\).

\[
\text{functor } f(x: \text{sig type } s \text{ type } t \text{ end}) = \text{struct datatype } w \text{ type } v = x.t \text{ end}
\]

\[
\text{let tyc } \hat{f} = \lambda \hat{x} :: \{ s :: k, t :: k \}.\{ w = \text{new}(0), v = \hat{x}.t \}
\]

\[
in \text{let } f = \Lambda \hat{x} :: \{ s :: k, t :: k \}.\lambda x : \{ \}.\{ \} \in \ldots
\]

For the above example functor \(f\), the primary tycon part is a tycon function \(\lambda \hat{x}.\{ w = \text{new}(0) \}\), which is exactly a straightforward translation of the structure entity expression for the functor body. The value part is \(\Lambda \hat{x} :: \{ s :: k, t :: k \}.\lambda x : \{ \}.\{ \}\). The secondary tycon part \(v = \hat{x}.t\) corresponds to the type definitions \texttt{type } v = x.t.

The above translation is sufficient in absence of value components, whose type annotations may refer to tycons (the type part of the functor). The value components in the functor parameter may refer to tycons in the functor parameter. The value components in the functor body may refer to tycons in either the functor parameter or body. However, since the tycons in the functor body are not accessible until the type part of the functor is applied, the value part of the functor must take an extra type parameter, the result of applying the type part of the functor.

\[
\text{functor } f(x: \text{sig type } s \text{ end}) = \text{struct datatype } w \text{ val } n : w \rightarrow w = \text{fn } z : w \Rightarrow z \text{ end}
\]
let tyc \( \hat{f} = \lambda \hat{x} :: \{ s :: k \}. \{ w = \text{new}(0) \} \)

\[ \text{in let } f = \Lambda \hat{x} :: \{ s :: k \}. \Lambda \hat{f}_{\text{res}} :: \{ w :: k \}. \lambda x :: \{}. \{ n = \lambda z : \hat{f}_{\text{res}}. w \} \text{ in ...} \]

The first curried type abstraction \( \Lambda \hat{x} :: \{ s :: k \} \) is the type part of the functor parameter. The second type abstraction \( \Lambda \hat{f}_{\text{res}} :: \{ w :: k \} \) is for the result of applying the type part \( \hat{f} \) to the same type part of the functor argument \( \hat{x} \). The type annotation in the body of the value part \( n \) can refer to the tycon \( w \) through \( \hat{f}_{\text{res}}. w \).

Functor application must coerce both the tycon and value parts of the argument structure.

\[ \text{structure } m = f(\text{struct datatype } s \text{ type } t = s \text{ end}) \]

\[ \text{let tyc } \hat{u} = \{ s' = \text{new}(0) \} \]
\[ \text{in let tyc } \hat{m} = \hat{f}(\{ s = \hat{u}.s', t = \hat{u}.s' \}) \]
\[ \text{in let } m = f[\{ s = \hat{u}.s', t = \hat{u}.s' \}](\{\}) \]
\[ \text{in ...} \]

In the above example, the translation first decomposes the argument structure into a primary tycon part \( \{ s' = \text{new}(0) \} \), a value part \( \{\} \), and a secondary tycon part \( t = \hat{u}.s' \). Note that the entity variable for the argument structure \( s' \) that is distinct from the entity variable in the formal parameter \( s \). The entity variable \( \hat{u} \) binds the raw, uncoerced primary tycon part of the argument structure. The raw, uncoerced secondary tycon part is directly inlined because the syntactic functor application inlines the argument structure. The tycon part of \( \hat{f} \) is applied to the coerced tycon part of the argument structure, \( \hat{f}(\{ s = \hat{u}.s', t = \hat{u}.s' \}) \) where \( \hat{f} \) here refers to the tycon function bound in the previous example. The value part of \( f \) is applied to the coerced tycon and value parts of the argument structure, \( f[\{ s = \hat{u}.s', t = \hat{u}.s' \}](\{\}) \).

Since the tycon part of functor is a translation of the functor entity expression, the tycon part also represents the nested functors in the functor body.

\[ \text{functor } f() = \text{struct functor } g() = \text{struct datatype } t \text{ end end} \]
let tyc \( \hat{f} = \lambda().\{\hat{g} = \lambda().\{t = \text{new}(0)\}\} \)

in let \( f = \Lambda() :: \{\}.\lambda() : \{\}.\{\} \)

The tycon part of the nested functor \( g \) is \( \lambda().\{t = \text{new}(0)\} \). Thus, the tycon part \( \hat{f} \) must be the first line in the example above. In this example, the value part is degenerate since there are no value components.

Formal functors are handled in the same way as above.

functor \( f(x:\text{sig functor } g(y:\text{sig type } t \text{ end}) : \text{sig type } v \text{ end end}) = \)

structure \( m = f(\text{struct functor } g(y:\text{sig type } t \text{ end}) = \text{struct type } v = y.t \text{ end end}) \)

The above example translates to the following \( F_\omega \) program.

let tyc \( \hat{f} = \lambda x :: \{\hat{g} :: \{t :: k\} \rightarrow \{v :: k\}\}.\{\} \)

in let \( f = \Lambda x :: \{\hat{g} :: \{t :: k\} \rightarrow \{v :: k\}\}.\lambda x : \{g : \forall \hat{g} :: \{t :: k\} \rightarrow \{v :: k\}\}.\{\} \)

in let tyc \( \hat{u} = \{\rho'_g = \lambda \rho'_y.\{\} \})

in let \( u = \{g' = \Lambda \hat{y}' :: \{\hat{g}' :: \{t' :: k\} \rightarrow \{v' :: k\}\}.\lambda \hat{y}'.\{\} \})

in let tyc \( \hat{m} = \hat{f}({\hat{g} = \hat{u}.\hat{g}'}) \)

in let \( m = f[{\hat{g} = \lambda \hat{y}'.\{\}},{\hat{v}' = \hat{y}'.t}](\{g = u.g'\}) \)

The arrow kinds are given to the tycon abstractions representing the formal functor, reflective of how functors can be viewed as type operators. The tycon and value parts of the uncoerced functor argument are \( \{\hat{g} = \lambda \hat{y}'.\{\}\} \) and \( \{g' = \lambda y'.\{\}\} \) respectively.

Support for type generativity requires some modification of the translation rules for functor declaration and functor application.
\( F = (\Pi \rho_x : \Sigma_{\text{par}} \Sigma', \langle \theta; \Upsilon^{\text{clo}} \rangle) \vdash \Sigma_{\text{par}} \leadsto \text{knd} \ k \quad \Upsilon_{\text{ins}} \vdash \Sigma_{\text{par}} \leadsto \text{type} \ t \)

\[ \text{strexp} \leadsto_{\text{exp}} e_\omega \]

**functor** \( f \langle \langle \rho, F, \Upsilon_{\text{ins}} \rangle \rangle (X : \text{sigexp}) = \text{strexp} \)

\( \leadsto_{\text{dec}} \rho =_{\text{tyc}} \theta, \rho = \Lambda \rho_x :: k. \lambda \rho_x : t. e_\omega \)

\[ \text{strexp} \leadsto_{\text{exp}} e_\omega \quad M_c \leadsto_{\text{tyc}} c \quad M_c = (\Sigma, R) \]

\( R \vdash \Sigma \leadsto_{\text{type}} lts_c \quad e_{\omega_c} = \text{coerce}(\rho_u, c, \{lts_c\}) \)

**structure** \( x \langle \langle \rho_x \rangle \rangle = p \langle \langle \bar{\rho}, \rho_{\text{par}} \rangle \rangle (\text{strexp}(\langle M_c, \rho_u \rangle)) \)

\( \leadsto_{\text{exp}} \text{let } \text{tyc } \rho_x = \bar{\rho}(\text{coerce}_{\text{tyc}}(\rho_u, c)) \text{ in let } \rho_x = \bar{\rho}(\text{coerce}_{\text{tyc}}(\rho_u, c)), \text{ let } \rho_u = e_{\omega_u} \text{ in } \bar{\rho}[c](e_{\omega_c}) \)

(8.60)
CHAPTER 9

CONCLUSION

The design of higher-order module systems is indeed rich and complex. Much of the richness is due to the introduction of functor actions and transparent signature matching semantics. While the $\beta$-reduction semantics for functor applications is both natural and intuitive, supporting such semantics in the presence of higher-order functors requires the richer static representation that the entity calculus affords. This present account improves the state-of-the-art in module system semantics by providing a formal account of true higher-order semantics and type generativity without resorting to re-elaboration of functor bodies.

9.1 Future Work

The entity calculus is a simple and natural way to represent module computation. It remains a question whether it can be easily adapted to support recursive modules while maintaining the strong normalization property.

The main argument that the design of the entity calculus makes is that functor actions should be calculated by the elaborator and not written down in a syntactic signatures. This kind of functor action inference suggests that inference may be able to play a larger role in module system semantics.

Definitional specifications such as definitional type specs and definitional structure specs are common in many implementations of the ML module system. For symmetry, it would seem natural that definitional functor specs would also be permitted. However, to the best of the author’s knowledge, this kind of definitional spec has not been investigated.
APPENDIX A
SML/NJ ELABORATION SEMANTICS

A.1 Signatures

\[ \Gamma, \Upsilon, C_{ep}, C_{sig} \vdash \text{sigexp} \Rightarrow_{\text{sig}} \Sigma \]

Signatures elaborate in the context of the following semantic objects:

**static environment** (\( \Gamma \)) The static environment interprets any nonlocal symbolic paths mapping them to semantic representations.

**entity environment** (\( \Upsilon \)) The entity environment interprets all external entity paths in datatype and datatype replication specifications and any other relativized type expressions.

**entity path context** (\( C_{ep} \)) The entity path context helps relativize types in where clauses, type, exception, and datatype/replication specifications. The elaborator also looks up the entity path context to obtain the entity path for structure definition specs. [so structure references are also relativized in certain circumstances] This is an auxiliary implementation mechanism used only in the process of building static representations – not part of the representations themselves.

**signature context** (\( C_{sig} \)) A signature context is a list of spec elements, specifically, structure spec elements. These spec elements are semantic representations, the product of elaborating earlier or outer scope syntactic specs. It plays a very specialized role. When elaborating datatype replication specs, the tycon of the datatype being replicated may be either relativized or unrelativized. If the tycon has been relativized, then the elaborator needs to look up the original tycon spec. The tycon spec must be either in the same or an outer context, both of which are recorded in the signature context. [seems slightly unexpected that the static environment can’t play this role.}
The signature elaboration process produces a semantic representation of a signature (\(\Sigma\) in fig. 7.2).

The main steps of signature elaboration are:

1. Elaborate outer where definitions – the ones applying directly to the current signature
2. Check for duplicate names
3. Relativize types for value, type, and data constructor specifications
4. Decorate static component specs with fresh entity variables
5. Push where definitions down to relevant type specification [does this process involve adjusting any relative references, i.e., entity paths?]

Signature elaboration proceeds by first eliminating where type constraints by translating the constraints into a list of relativized semantic definitional type constructors. This list of type constructors is put aside until the very end of signature elaboration. At that point, the elaborator goes through all the specs in the signature and applies all applicable definitional constraints. In SML/NJ, definitional constraints only apply to open types and datatypes. The precludes the possibility of overconstraining a tycon with multiple potentially contradictory definitions such as the following:

\[
\text{signature } S = \text{sig type } t = \text{int end where type } t = \text{bool}
\]

However, observe that not all overconstraining by type definitions and where type results in contradiction. I describe a policy that relaxes the semantics of where type to permit definitional type names on the left hand side in a later section.

For functor signatures, the elaborator first processes the parameter signature and then elaborates the functor body signature using a static environment extended with a binding mapping the parameter’s symbolic name to a full signature where the free instantiation realization which will be subsequently ignored. The resultant body signature and parameter
signature with parameter entity variable are put together as the semantic representation of the functor signature.

When elaborating structure specs, we have a special case. In order to elaborate any specs that may follow a structure spec, we need to extend the static environment with a binding for the structure spec. Unfortunately, we do not have a full signature for that structure spec because there is no realization. Hence, the elaborator creates a placeholder free instantiation as in the case above.

During signature elaboration, the elaboration of a spec may require the result of the elaboration of previous specs. The notation $\Gamma[\Sigma_1]$ puts the result of the elaboration of the preceding specs in the static environment by dropping the entity variables because static environment bindings do not include entity variables.

The rebind function rebinds q to a new spec in the semantic signature.

### A.2 Functor Signatures

\[
\Gamma, \Upsilon, C_{ep}, C_{sig} \vdash fsgexp \Rightarrow fsg S^f
\]

Functor signatures elaborate in a context of:

- **static environment**
- **entity environment**
- **signature context**
- **entity path context**

Elaboration produces a semantic representation of a functor signature.

The main steps are:

1. Elaborate the parameter signature expression
2. Extend the static environment with a binding from parameter name to parameter signature. This binding is the same as the special binding form used for structure specs.

3. Extend the signature context with a binding from parameter name to decorated structure spec (with a fresh entity variable for the functor parameter) corresponding to the parameter structure. The elaborator needs this decorated structure spec because the body of the functor may contain datatype replication specs that replicate some datatype whose tycon is declared in the parameter.

4. Elaborate the body signature expression

\[ \Sigma \vdash \sigma \uparrow \Upsilon' \]

Signature instantiation produces a free instantiation of the given signature \( \Sigma \) with the least amount of type sharing necessary to satisfy the signature.

\section{A.3 Declarations}

\[ \Gamma, \Upsilon, C_{ep}, C_{elab} \vdash decl \Rightarrow decl (\eta, \Gamma', \Upsilon') \]

Elaboration of declaration uses the following context:

\textbf{static environment} Elaboration augments static environments with new bindings corresponding to the declarations. It also uses them to interpret external symbolic paths.

\textbf{entity environment} The elaborator builds up an entity environment for structures inside of functors.

\textbf{elaboration context} \( C_{elab} \) The elaboration context distinguishes between top level, in structure, in functor, and in signature contexts. The primary distinction is whether
elaboration is occurring inside a functor or not. When inside of a functor, entity declarations must be produced whereas they are unnecessary for other contexts. The point when the elaborator first enters into a functor also partitions externally-defined type constructors which are rigid and locally-defined type constructors which are flexible.

**entity path context** The entity path context is used to relativize entity paths.

The elaboration process yields the following principal results:

- **typed abstract syntax**
- **entity declaration**
- **static environment**
- **entity environment**

### A.3.1 Structure Declaration

The main steps of structure declaration elaborations are:

1. elaborate signature expressions if any
2. elaborate structure definiens expression
3. signature matching
4. build entity environment from entity environments produced by first two steps
5. build typed abstract syntax from coercion
6. add binding to full signature to static environment

A structure declaration consists of a name, constraining signature expression, and definiens expression. The elaborator first elaborates both the signature expression and the structure
definition (a structure expression). Then the elaborator does signature matching on the resultant semantic signature, full signature, typed abstract syntax, and entity expression for the structure to produce a typed abstract syntax declaration, full signature, and entity expression for the coerced structure. The entity environment delta from elaborating the structure does not play a role in signature matching.

If a signature constraint is present, the entity environment delta from the structure will need to be extended with a binding that maps an entity variable for the short-lived temporary binding of the uncoerced structure. This new entity environment will be combined with a binding for entity variable for the coerced realization and the contextual entity environment to yield the final result entity environment.

The full signature of the possibly coerced structure is combined with the typed abstract syntax to produce a structure binding in typed abstract syntax. The principal results of structure declaration elaboration include this typed abstract syntax, a static environment extended with a binding for the relativized coerced full signature, the final result entity environment, and an entity declaration mapping the entity variable for the coerced structure to the entity expression for the same.

[Idea: The only difference between the bindStr in the STRB and the resStr in the LETstr part of the STRB is the dynamic access. Couldn’t bindStr be used in both places?] [Idea: The cruft of ascription handling in elabStrbs is likely due to the lingering presence of AbsDec along with the currently used forms of Transparent and Opaque constraints. Eliminate AbsDec and things may simplify. ] [Idea: The elaboration code for signature constraints is duplicated in the elabStr[ConstrainedStr] and elabStrbs case. This duplication does not appear necessary. Can’t ConstrainedStr be expanded into a structure binding or vice versa?]

\[ \text{A.3.2 Signature Declaration} \]

The main steps of signature declaration elaboration are:
1. elaborate signature expression

2. instantiate resultant signature

3. add signature to static environment

For signature declarations which can only occur at the top level, the elaborator may attempt to instantiate the elaborated signature to serve as a sanity check. This check is not necessary for correctness, but it does eliminate certain uninhabitable signatures at elaboration-time. [Idea: Why don’t we do something similar for functor signatures?]

### A.3.3 Functor Declaration

1. elaborate functor

2. build typed abstract syntax

3. extend entity environment with entity declaration from functor elaboration if in functor according to elaboration context

4. add full functor signature to the static environment

Upon elaboration of a functor declaration, the elaborator must extend the entity environment with the delta produced during elaboration of the functor and a binding for the functor entity. Furthermore, the functor entity declaration must be produced. If this functor declaration is not nested, the extended entity environment and entity declaration are unnecessary.

[Idea: Currying value-level functions has definite benefits in terms of partial evaluation. Is currying functor parameters as prevalent and if so does it carry the same benefits?]
A.4 Structures

\[ \Gamma, \Upsilon, C_{ep}, C_{elab} \vdash strexp \Rightarrow str (M, \varphi, \Upsilon') \]

Structures elaborate in the following context:

static environment

entity environment

elaboration context The explanation is in the above section.

entity path context The elaborator will look up functors (as in functor application) in the entity path context. The resultant entity path will be used to fill out the entity expression for the functor application. Structure variables will also be looked up in the entity path context to translate it into a corresponding entity expression.

The principal results are:

typed abstract syntax Binds a symbolic name to the possibly coerced full signature and a body comprised of a let-expression with the typed abstract syntax from the uncoerced structure and the bindings extracted from the static environment

full signature (M) The full signature is the principal product of structure expression elaboration. It gives the static description of the structure expression.

entity expression This entity expression is used to form the entity declaration in the case where the above structure expression is found inside of another functor’s body, thus necessitating recording of the entity function.

entity environment
A.4.1 Base Form

The main steps of base structure expression elaboration are the following:

1. elaborate declarations

2. extract full signature from resultant static environment

3. combine entity environments from first two steps

4. build typed abstract syntax

The elaborator deals with the basic form of a structure by elaborating the constituent declarations and extracting the signature the resultant static environment. The elaborator can infer or extract a full signature from the static environment derived during elaboration of a structure. Extraction forms a list of specifications such that each value, data constructor, structure, functor, and type constructor binding in the static environment has a corresponding specification. Entity variables for each specification is obtained by looking up the entity path context.

Extraction computes the specifications from the full signatures of the structures and functors. Type constructor specifications are minimal. They only mention the name and arity of the said type constructor, thus are opaque.

More interestingly, extraction must also build entity declarations and entity environments to go along with each specification. The entity declarations are simply declarations that bind the a fresh entity variable to the entity path of the static entity. The result entity environments bind this fresh entity variable to the realization. All structure, functor, and value bindings are also returned as part of the result.

The entity environments and declarations produced by declaration elaboration and signature extraction are combined with the contextual entity environment and used to form the final structure realization. The elaborator must concatenate the entity environment from
declaration elaboration onto the result entity environment because of possible inexplicitness. The extracted signature and realization are combined to form the full signature. The entity declarations from elaboration and extraction are sequenced together to form the final structure entity expression. The typed abstract syntax from declaration elaboration and the bindings from extraction and the static environment bindings returned from extraction are combined to form the typed abstract syntax for this structure.

A.4.2 Signature Extraction

\[ C_{ep}, C_{elab} \vdash \Gamma \hookrightarrow (M, \eta) \]

Signature extraction requires the following environments:

**static environment** The static environment is decomposed into its bindings from which the elaborator builds a signature.

**entity path context** The entity path context is used to look up the entity paths (i.e., the relativized entity paths) for each binding in the static environment. It will also be used to relativize entity paths.

**elaboration context** \( C_{elab} \) Types will be relativized only if elaboration context is in functor.

The process produces the following:

**spec elements** \( \text{elems} \) specs produced by relativizing types in static environment bindings and reverse lookup of structures in the entity path context

**static environment bindings** \( \text{bindings} \) the list keeps only the structure, functor, tycon, value, and data constructor bindings. These bindings will become the let-body in the typed abstract syntax.
**entity declaration** ($\eta$)  This entity declaration contains an entity declaration for each static (i.e., non-value) spec. The entity declaration is either a variable entity or a constant entity depending on whether there is a corresponding entity path in the entity path context. They are propagated directly through elaboration.

**entity environment** ($\Upsilon'$) The entity environment, which will be the extracted (inferred) structure realization, is extended with the entities from structure, functor, and tycon bindings.

When signature extraction looks at an entity, there are two main cases:

**tycon** Volatile tycons have stamps which uniquely identify them in the static environment. By exhaustively scanning through the static environment, signature extraction builds up the entity path leading up to the binding occurrence (not applied occurrence) of that entity. The binding occurrence of the entity is unique in a module system without sharing type constraints. In the presence of sharing type constraints, the canonical entity path for an entity can be defined as the path leading up to the first binding occurrence where first is first in terms of syntactic ordering.

**structure and functor** Structure and functor entities must be extended with its binding occurrence entity variable. This entity variable will identify the associated spec in the signature and by extension help calculate the entity path.

When signature extraction encounters an entity that entity may or may not be already in the entity environment. If it is already there, then

Note that entity environments are not injective during entity evaluation because entity paths will be looked up and another entity variable may now be mapped to that entity. Because these entity environments are ultimately used during elaboration when propagated through by functor application elaboration, this means that the entity environment may be non-injective throughout elaboration.
functor F(X:sig type t end) =
struct
  type u = X.t
end

Lemma 16

The entity environment is injective during elaboration.

Proof: Entity environments are constructed by signature instantiation, structure expression elaboration, signature extraction, functor elaboration, signature matching, and entity evaluation.

During entity evaluation Entity evaluation constructs new entity environments by binding fresh entity variables to the result of entity evaluation. Will an entity declaration or expression ever evaluate to an entity that already occurs in the entity environment? Yes, evaluating an entity path will look it up in the entity environment thereby yielding an entity that already exists in the environment.

Lemma 17

There exists a retraction of the entity environment lookup function.

Proof: This follows from lem. 16.

Static environment bindings contain tycons, signatures, and functor signatures necessary to reconstitute the specs in a signature. Given a static environment binding for a static entity, the corresponding entry in the entity path context gives the relativized entity path. If the relativized entity path is a single entity variable, that entity variable is the one for this signature spec. Otherwise, a fresh entity variable is used for the spec. In this latter case,
the entity declaration will either be equal to the realization of the entity or an entity path depending on whether the entity is in the entity path context.

Signature and functor signature bindings are not permitted inside of structures so signature synthesis ignores them.

\section*{A.4.3 Functor Application}

\[ \Gamma, C_{ep} \vdash \theta F(\varphi M) \rightsquigarrow (M_{res}, \varphi_{res}) \]

Functor application requires the following:

**static environment** (\(\Gamma\)) The static environment is used to signature match the argument full signature to the formal parameter signature.

**entity path context** (\(C_{ep}\)) The entity path context is used as the evaluation entity environment when evaluating the application of the entity function.

**full functor signature** (\(F\)) This is the full functor signature for the functor in the application.

**functor entity expression** (\(\theta\)) This is the functor entity expression for the functor in the application. This will be used to constructor the application structure entity expression.

**full signature** (\(M\)) This is the full signature for the argument structure.

**structure entity expression** (\(\varphi\)) This is the structure entity expression for the argument structure. This entity expression will be coerced by signature matching, the result of which will be used to construct the application structure entity expression.

The process results in:

**full signature** (\(M_{res}\)) This is the full signature of the functor application result.
entity expression \((\varphi_{res})\) This is the application structure entity expression, memoizing all latent functor actions.

The main steps of functor application elaboration are:

1. look up functor in static environment

2. elaborate argument structure expression

3. extend entity environment with uncoerced structure realization

4. signature match uncoerced argument with formal functor parameter signature

5. evaluate functor body entity expression with coerced argument realization

After elaborating the argument structure expression and functor, the elaborator looks up the entity path for the functor to form either a functor variable or constant entity expression. [Can constant be eliminated?] Then, the elaborator does signature matching on the functor formal parameter signature and the actual argument structure using the closure entity environment in the functor realization. The elaborator then evaluates the functor body replacing the formal parameter with the coerced argument realization. The resultant evaluated functor body realization is combined with the functor body signature to form the full signature of the result. The coerced argument full signature is also used to produce the usual typed abstract syntax and structure entity expression, an apply.

The semantics for static functor application varies considerable among ML variants. Leroy’s semantics only permits applying functors to paths, thus requiring all functor arguments to be A-normalized. DCH does not impose coercive signature matching in the functor argument. The key distinction of GEN’s semantics is the judgment \(\vdash \varphi_{app} \Downarrow \Upsilon_{app}\). This judgment evaluates only the static content of the functor application. In DCH, the argument structure is substituted directly into the result signature, thus requiring a complex theory of projectibility.
A.5 Functors

Functor elaboration assumes a context consisting of:

- static environment
- entity environment
- elaboration context
- entity path context

and produces four principal results

- typed abstract syntax comprised of a functor binding containing a copy of the full functor signature and the typed abstract syntax for a functor containing the full signature for the free instantiation of the parameter, the full signature of the functor body, and the typed abstract syntax for the body

- full functor signature (F)
- functor entity expression
- delta entity environment

A.5.1 Functor Variable

The main steps of functor variable elaboration are:

1. look up symbolic path to functor in static environment to get full functor signature
2. look up functor in entity path context to get relativized entity path
3. if there is an ascribed functor signature, elaborate the functor signature and signature match full functor signature and ascribed functor signature to yield the coercion
### A.5.2 Base Functor

The main steps of functor elaboration are:

1. elaborate the formal parameter signature expression

2. instantiate formal parameter signature

3. extend entity environment and static environment by binding parameter expression and parameter’s full signature

4. elaborate functor result signature

5. elaborate functor body using the environments from step 3

6. signature match functor body with functor body signature

7. build functor entity expression, full functor signature, and typed abstract syntax

Functors elaborate by elaborating the parameter signature expression and instantiating the resultant semantic parameter signature. The elaborator then elaborates the functor body using an entity environment extended with the resultant free instantiation of the parameter signature, a static environment extended with the full signature, an elaboration context that differentiates between type constructors from inside and outside of any/all functors marking those type constructors as volatile, and an entity path context that is extended with the entity variable for the functor parameter. The entity expression from the functor body is used to form the functor entity expression \((\lambda_{\rho_{\text{param.bodyExp}}})\). The full signature from the functor body is used to form the full functor signature. The functor realization’s closure environment is the current context entity environment.

Some functors are themselves a result of functor application. The elaborator treats these by expanding the syntax tree to a let-expression where the only definition is a structure
application and the body is a functor variable that extracts out the functor component of the definition.

For functor-level let-expressions, the typed abstract syntax of the elaborated definition (a declaration) and body (a functor) are sequenced. The entity declarations are collected in a let entity expression. The resultant entity environments are also concatenated.

Functor variables may include a transparent ascription in which case the functor signature must be elaborated and the elaborator must match the functor with the functor signature possibly yielding a coercion expression.

## A.6 Signature Matching

### A.6.1 Structure Signature Matching

\[ \Gamma, \Upsilon \vdash \varphi M \preceq \Sigma \Rightarrow (M_{\text{coerced}}, \varphi_{\text{coerced}}) \]

Structure signature matching involves the following:

- **semantic signature** \((\Sigma)\) the ascribed signature to be matched against
- **full signature** \((M)\) the full signature of the structure to be matched
- **structure entity expression** \((\varphi)\)
- **entity environment** \((\Upsilon)\)
- **static environment** \((\Gamma)\)

The process produces the following results:

- **full signature** \((M')\) the full signature of the coerced structure
- **structure entity expression** \((\varphi')\) an entity expression representing the coerced structure.

The steps to structure signature matching is quite substantial:
1. match the elements in the ascribed signature

   (a) get the element’s entity variable from the signature

   (b) look up that entity variable in the full signature of the uncoerced structure to get
       the realization for that element

   (c) match the spec with the realization

   (d) produce the thinned representations

2. check that sharing constraints are followed in the result entity environment

3. build the full signature for the coerced structure (signature is the ascribed signature
   and realization is the result entity environment)

4. build typed abstract syntax that binds symbolic structure name to full signature and
   a let-expression with coerced typed abstract syntax declarations and bindings from
   matching the elements

For structure specs, the matching requires the following steps:

1. look up the element’s entity variable and the uncoerced’s structure’s full signature as
   above

2. look up the realization for any structure definitional spec and match those with the
   uncoerced structure

3. match the uncoerced structure’s full signature with the structure spec

4. extend the entity environment with the coerced realization

5. produce an entity declaration with the entity expression from the coercion

6. return this extended entity environment, this entity declaration, the binding from the
   coercion, and the typed abstract syntax from the coercion
For functor specs:

1. look up the element’s entity variable and full functor signature

2. match the functor spec with the full functor signature noting the entity path to the functor element

For value specs:

1. look up the symbolic name of the spec in the signature part of the full signature to get the corresponding uncoerced spec (i.e., the actual type)

2. match the types according to core language typing discipline

3. record any instantiations due to the coercion from the core language type match in the typed abstract syntax and bindings

A.6.2 Functor signature matching

\[
\Gamma, \Upsilon \vdash F \preceq S^f \Rightarrow (F_{\text{coerced}}, \theta_{\text{coerced}})
\]

Functor signature matching requires:

**functor signature** \( (S^f) \) the ascribed functor signature (the “spec”)

**full functor signature** \( (F) \) the full functor signature for the functor to be coerced

**entity environment** \( (\Upsilon) \)

**static environment** \( (\Gamma) \)

The process produces the following results:

**full functor signature** \( (F_{\text{coerced}}) \) the coerced full functor signature

**functor entity expression** \( (\theta_{\text{coerced}}) \) the coerced functor entity expression
Figure A.1: Normalized module calculus NRC

1. Instantiate the formal parameter signature from the spec and given entity environment

2. Take that realization and form a full signature for the spec formal parameter

3. Evaluate the functor application of the actual functor to the spec full signature

4. Match the full signature from the application result to the specified functor result signature using the entity expression from the evaluation and entity environment extended with the spec formal parameter signature’s full signature (from 1)

5. build the coerced functor realization using the given entity environment (not the extended one) a let-expression with the uncoerced functor realization as the definiens and entity expression from the coercion in the previous step as the body.

6. return the coerced full functor signature
The abstract approach relies on signature subsumption and strengthening to propagate types for functor application.

A.7.2 Shao

A.7.3 Parameterized Signatures (Jones)

Jones proposed using signatures parameterized on type constructors to express type propagation[33].

Under this system, the apply functor is expressed as the following:

```plaintext
signature S t = sig ... end
functor apply(functor f(x:S t):S u
structure a:S t): S u
  = f a
```

This proposal argues removing type components from structures and wholly relying on type constructor parameters in these parameterized signatures to express type relationships. The resulting system only requires higher-order and first-class polymorphism and records, which encode the value-only structures. First-class polymorphism permits what amounts to functor application over first-class structures. Jones’ approach is similar to Shao’s in that it uses higher-order type constructors to encode type sharing. Under this scheme, all types must be declared at the top level. This idea follows Harper, Mitchell, and Moggi’s
observation of phase splitting\[^{30}\]. The difference, however, is that Jones proposes such a separation of type and value modules in the surface language.

The drawback in this approach is that the programmer will have to know \emph{a priori} the types that must be propagated. Jones’ approach separates type abstraction concerns from the module system. However, this means that the scope of such type parameters are no longer restricted. Even when type components in modules are once again permitted, expressing the correct type sharing will force the programmer to lift type components to the top level, thus negating the advantage of the module system in favor of a simpler type-theoretic explanation.

This technique does not support more general forms of functor actions such as generative datatypes though Jones suggests using core language existentials or abstypes.

\textit{A.7.4 Dynamic ML}

Dynamic ML suggests the addition of a where type definition (where datatype) that shares a datatype and its constructors such that no new datatype is generated \[^{25}\]. The where datatype definition can be derived from the composition of datatype replication and a regular where type definition both at the signature level.

\begin{verbatim}
functor F(structure M : SIG where datatype t = A of int | B of bool)
functor F(datatype t = datatype t structure M : SIG where type t = t)
\end{verbatim}

This encoding is incomplete. The replicated datatype in the functor parameter must be hidden so that only M.t remains. Furthermore, the replicated data constructors are in the functor parameter but not M.

\textit{A.8 Comparison with SML/NJ}

Currying adds a minor complication that can be encoded by expanding the functor out to a nested functor signature.
Datatype Replication Specs

Datatype replication is treated specially. The tycon which is being replicated may occur in a few contexts:

functor parameter

\[
\text{functor } F(X:\text{sig datatype } t = A \text{ end} ) = \\
\text{struct} \\
\text{structure } M : \text{sig datatype } t = \text{datatype } X.t \text{ end} \\
\text{end}
\]

earlier spec

\[
\text{sig} \\
\text{datatype } t = A \\
\text{datatype } u = \text{datatype } t \\
\text{end}
\]

outer scope

\[
\text{sig} \\
\text{structure } M : \\
\text{sig} \\
\text{datatype } t = A \\
\text{structure } N : \\
\text{sig} \\
\text{datatype } u = \text{datatype } t \\
\text{end} \\
\text{end} \\
\text{end}
\]

The source tycon, the tycon that is being replicated, must be either a generative tycon of kind datatype or a relativized tycon. In the generative tycon case, the elaborator will relativize the tycon, wrap the resulting relativized tycon in a definitional tycon, and add the data constructor specs to the cumulative elements list. To elaborate the remainder of the signature specs, the static environment is extended with a new relativized tycon for this new tycon.
In the relativized case (that is the target tycon was relativized), the elaborator looks up the entity path in the signature context to obtain the datatype family. From the datatype family, the elaborator extracts the member of interest and constructs a definitional tycon around the original relativized target tycon. This definitional tycon is added to the elaborated elements list. A new relativized tycon (for this replication) is added to the static environment. The elaborator then adds the data constructor specs, checking that each relativized parameter tycon is bound in the entity environment.
APPENDIX B
IMPLEMENTATION IN SML/NJ

The semantics in this dissertation departs from the implementation (as of SML/NJ 110.70) in SML/NJ in a number of significant ways. Whereas the THO semantics constructs entity expressions for all module expressions, SML/NJ distinguishes between expressions based on whether they are inside a functor. For those expressions outside of a functor, no entity expression needs to be constructed since all tycons including datatypes are nonvolatile because they have a single, fixed instantiation. SML/NJ’s distinction between volatile and non-volatile contexts (i.e., in functor or outside) is an optimization that skips the construction of entity expressions during elaboration of all expressions outside of any functor.

A major difference is how relativization is performed. SML/NJ uses a construct called the entity path context to memoize the inverse lookup of the entity environment for the canonical entity path to a given entity. The entity path context must be threaded through the entire elaboration process and complicates the elaboration process.

For structure entities, SML/NJ assigns each a stamp to optimize equality checks. In SML90, this stamp was used to implement structure sharing, but the post-SML97 versions only use the stamp as a shortcut during typechecking. Because SML/NJ supports sharing type constraints, signature instantiation is more involved. It must resolve all sharing, which it does using an algorithm adapted from the Patterson-Wegman linear unification algorithm [56].
APPENDIX C

SEMANTICS PROOFS

C.1 Type System

Lemma 18 (Monotypes are strongly normalizing)
If $C^s \Downarrow C^s_1$ and $C^s \Downarrow C^s_2$, then $C^s_1 = C^s_2$ and $C^s$ is a $C^{nf}$.

Proof: There is a precise embedding of the monotype language in the simply-typed $\lambda$-calculus. Because the simply-typed $\lambda$-calculus is strongly normalizing, so this the monotype language. ■

Lemma 19 (Inversion)
1. If $\Delta \vdash \alpha : \Omega$, then $\alpha \in \Delta$.

2. If $\Delta \vdash \text{C}_\lambda(\vec{C}_s) : \Omega$, then $\Delta \vdash \text{C}_\lambda : \Omega^n \Rightarrow \Omega, |\vec{C}_s| = n$, and $\Delta \vdash \text{C}_s_i : \Omega$ for all $i \in [1, n]$.

3. If $\Delta \vdash \lambda \vec{\alpha}.\text{C}_s : \Omega^n \Rightarrow \Omega$, then $\Delta[\alpha_1] \ldots [\alpha_n] \vdash \text{C}_s : \Omega$.

4. If $\Delta \vdash \text{typ}(\text{C}_s) : \Omega$, then $\Delta \vdash \text{C}_s : \Omega$.

5. If $\Delta \vdash \forall \vec{\alpha}.\text{C}_s : \Omega$, then $\Delta[\alpha_1] \ldots [\alpha_n] \vdash \text{C}_s : \Omega$.

Proof: By syntax-directedness of kind system and inspection ■

Lemma 20 (Kind Preservation)
If $\emptyset_{knds} \vdash \text{C}_s : \Omega$ and $\text{C}_s \Downarrow_{tyc} \text{C}^{nf}$, then $\emptyset_{knds} \vdash \text{C}^{nf} : \Omega$.

Proof: [by induction on the typing derivation] The interesting case is the application rule (4.7). By inversion $\emptyset_{knds} \vdash \text{C}_\lambda : \Omega^n \Rightarrow \Omega$ and $\emptyset_{knds} \vdash \text{C}_s_i : \Omega \forall i \in [1, n]$ (0). By definition, $\text{C}_\lambda$ must be following two cases:
\[ \lambda \vec{\alpha} \mathcal{C}^{\prime} \] Then by the \( \beta \)-reduction rule, \( \mathcal{C}^s \downarrow_{tyc} \mathcal{C}^{nf}_i \) (1) and \( \mathcal{C}^s \downarrow_{tyc} \mathcal{C}^{rs}(\mathcal{C}^{nf}_{/\vec{\alpha}}) \) (2). By induction for (0) and (1), \( \emptyset_{knds} \vdash \mathcal{C}^{nf}_i : \Omega \) for all \( i \in [1, n] \) (3). If \( \mathcal{C}^{rs} = \alpha \), then done by (3). If \( \mathcal{C}^{rs} \) is an application, then done by induction.

\[ \tau^n \text{ Vacuously true} \]

Lemma 21 (Progress)

If \( \emptyset_{knds} \vdash \mathcal{C}^s : \Omega \), then \( \mathcal{C}^s \downarrow \mathcal{C}^{nf} \).

Proof: [by induction on the typing derivation]

C.2 Entity Calculus

Lemma 22 (Extended Entity Environment Lookup Terminates)

\( \Upsilon(\vec{\rho}) \) terminates.

Proof: [by induction on the length of \( \vec{\rho} \)]

Lemma 23 (Entity evaluation terminates)

The following evaluation derivations terminate (has a finite derivation):

1. \( \Upsilon \vdash \varphi \downarrow_{str} R \)

2. \( \Upsilon \vdash \eta \downarrow_{decl} \Upsilon' \)

3. \( \Upsilon \vdash \theta \downarrow_{fct} \psi \)

Proof: [by the size of the entity expression (excluding the closure environments) and entity environment]
1. **Rule 6.1** This follows from lem. 22.

**Rule 6.2** This follows from (2) by induction.

**Rule 6.3** The entity expression calculus is a simply-typed $\lambda$ (which known to be strongly normalizing) where types are implicit.

By induction (3), the functor entity evaluation terminates. By induction (1), the argument structure entity evaluation terminates. The functor entity $\psi = \langle \rho_{arg}.\varphi'; \Upsilon^{clo} \rangle$ must have either been in the entity environment $\Upsilon$ or be $\langle \theta; \Upsilon \rangle$.

Because either $|\Upsilon^{clo}| < |\Upsilon|$ or $|\Upsilon^{clo}| = |\Upsilon|$ for the respective cases above, $|\Upsilon^{clo}| \leq |\Upsilon|$. By assumption, $|R_{arg}| < |\varphi| + |\Upsilon|$.

If the former, then $|\psi| < |\Upsilon| < |\theta(\varphi)| + |\Upsilon|$. By induction (1), the evaluation of $\varphi'$ must terminate.

If the latter, then $|\psi| = |\theta| + |\Upsilon| < |\theta| + |\Upsilon| + |\varphi|$. By induction (1), the evaluation of $\varphi'$ must terminate.

**Rule 6.4** By induction (3), the functor entity evaluation terminates. By induction (1), the argument structure entity evaluation terminates. By lem. 38, instantiation terminates.

**Rule 6.5** By induction (2), entity declaration terminates. By induction (1), the evaluation of the body of the let terminates. Therefore the whole rule terminates.

2. **Rule 6.6** Trivial because no premises

**Rule 6.9** By induction (2)

**Rule 6.7** By induction (1) and (2)

**Rule 6.8** By induction (1) and (2)

3. For functor entity expressions, this is trivial. By inspection, each of the rules does not depend on any premise with the sole exception of entity environment lookup, which is
known to terminate by lem. 22. They are single-step rules that always apply and will terminate.

Lemma 24
If $\Upsilon \vdash \varphi \downarrow_{\text{str}} R$, then $\Upsilon \subseteq R$.
If $\Upsilon \vdash \eta \downarrow_{\text{decl}} \Upsilon'$, then $\Upsilon \subseteq \Upsilon'$.
If $\Upsilon \vdash \theta \downarrow_{\text{fct}} \langle \theta; \Upsilon' \rangle$, then $\Upsilon \subseteq \Upsilon'$.

Proof: [by inspection of rules]

C.3 Elaboration

C.3.1 Type Elaboration

Lemma 25 (Monotype Elaboration Preserves Kinding)
If $\Gamma, \Delta \vdash C^s : \Omega$ and $\Gamma, \Upsilon \vdash C^s \Rightarrow_{\text{mt}} \mathcal{C}^s$, then $\Delta \vdash \mathcal{C}^s : \Omega$.

Proof: [by induction on the derivation of $\Gamma, \Delta \vdash C^s : \Omega$]

Rule 4.1 (tyvar) $C^s = \alpha$, therefore rule 7.1 is the only applicable elaboration rule. By inversion of the syntactic type kinding, $\alpha \in \Delta$ so $\Delta \vdash \alpha : \Omega$ by rule 4.6.

Rule 4.2 (app) $C^s = p(\tilde{C}^s)$, therefore rule 7.2 is the only applicable elaboration rule. Let $n = |\tilde{C}^s|$. By inversion of the syntactic type kinding, $\Gamma, \Delta \vdash \tilde{C}^s_i : \Omega \forall i \in [1, n]$. By inversion of rule 4.2, $\Gamma, \Upsilon \vdash C^s_i \Rightarrow_{\text{mt}} \mathcal{C}^s_i \forall i \in [1, n]$. By induction, $\Delta \vdash \mathcal{C}^s_i : \Omega \forall i \in [1, n]$. 

147
Lemma 26 (Tycon Elaboration Preserves Kinding)

If $\Gamma, \Delta \vdash C^{\lambda} : \Omega^n \Rightarrow \Omega$ and $\Gamma, \Upsilon \vdash C^{\lambda} \Rightarrow_{tyc} C^{\lambda}, \Delta \vdash C^{\lambda} : \Omega^n \Rightarrow \Omega$.

**Proof:** This follows directly from lem. 25.

---

C.3.2 Signature Elaboration

Lemma 27

If $AT(\Sigma) = \emptyset$, for all $\Sigma_i \in \Gamma. AT(\Sigma_i) = \emptyset$, and $\Gamma, \Upsilon, \Sigma \vdash sigexp \Rightarrow_{sig} \Sigma'$, then $AT(\Sigma') = \emptyset$.

**Proof:** [by induction on the derivation]

**Rule 7.6** By assumption, $AT(\Gamma(x)) = \emptyset$.

**Rule 7.7** By induction, $AT(\Sigma') = \emptyset$. By definition of $C^{\lambda}$, $AT(C^{\lambda}) = \emptyset$.

**Rule 7.8** This depends on the above property holding for $\Gamma, \Upsilon, \Sigma \vdash spec \Rightarrow_{spec} \Sigma'$. The open tycon case is trivial. The type and val spec cases are true by definition. The structure and functor cases are by induction.

---

Lemma 28

If $\Gamma \vdash C^{\lambda} \Rightarrow_{tyc} C^{\lambda}$, then $AT(C^{\lambda}) \subseteq AT(\Gamma)$.

Lemma 29

If $AT(\Gamma) \subseteq AT(\Upsilon)$ and $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} ((\Sigma, R), \varphi)$, then $AT(M) \subseteq AT(R)$.

Lemma 30

If $AT(\Gamma) \subseteq AT(\Upsilon)$ and $\Gamma, \Upsilon, \emptyset_{sig} \vdash sigexp \Rightarrow_{sig} \Sigma_x$, then $AT(\Sigma_x) \subseteq AT(\Upsilon)$.

Lemma 31

If $AT(\Gamma) \subseteq AT(\Upsilon)$ and $\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$, then $AT(\Gamma') \subseteq AT(\Upsilon')$.  

148
Proof: [by induction on the derivation of $\Gamma, \Upsilon \vdash d^m \Rightarrow_{decl} (\eta, \Gamma', \Upsilon')$]

Rule 7.25 (empty) $AT(\emptyset_{se}) \subseteq AT(\emptyset_{ee})$.

Rule 7.26 (val) Requires $AT(\emptyset) \subseteq AT(\Gamma)$. Because $\Rightarrow_{core}$ cannot generate new atomic tycons, $AT(\emptyset) \subseteq AT(\Gamma) \cup AT(\Upsilon) \subseteq AT(\Upsilon)$ by assumption. By induction, $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.27 (typedef) By lem. 28, $AT(\emptyset \lambda) \subseteq AT(\Gamma)$. Therefore $AT(\emptyset \lambda) \subseteq AT(\Upsilon)$. By induction, $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.28 datatype By assumption $AT(\Gamma) \subseteq AT(\Upsilon)$. By extension, $AT(\Gamma[t \mapsto \tau^n]) \subseteq AT(\Upsilon[\rho_t \mapsto \tau^n])$. By induction $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.29 structure By lem. 29, $AT(M) \subseteq AT(R)$. Thus, $AT(\Gamma[X \mapsto (\rho, M)]) \subseteq AT(\Upsilon[\rho \mapsto R])$. By induction, $AT(\Gamma') \subseteq AT(\Upsilon')$.

Rule 7.30 functor By lem. 30, $AT(\Sigma_x) \subseteq AT(\Upsilon)$. Certainly, $AT(\Gamma[X \mapsto (\rho, (\Sigma_x, R_x)])] \subseteq AT(\Upsilon[\rho_x \mapsto R_x])$. All tycons in $\Sigma_{res}$ are relativized so $AT(\Sigma_{res}) = \emptyset$. Therefore $AT(\Gamma[f \mapsto (\rho, (\Pi \rho_x : \Sigma_x. \Sigma_{res}, \psi)])] \subseteq AT(\Upsilon[\rho \mapsto \psi])$. By induction, $AT(\Gamma'') \subseteq AT(\Upsilon'')$.

Lemma 32

If $\Gamma, \Upsilon \vdash strexp \Rightarrow_{str} ((\Sigma, R), \varphi)$ and $EV(\Gamma) \subseteq dom(\Upsilon)$, then $EV(\Sigma, R) \subseteq dom(R)$ and $EV(\varphi) \subseteq dom(R)$.

Lemma 33

If $\Gamma, \Upsilon, \emptyset_{sig} \vdash sigexp \Rightarrow_{sig} \Sigma$ and $EV(\Gamma) \subseteq dom(\Upsilon)$, then $EV(\Sigma) \subseteq dom(\Upsilon)$.

Definition 4

$\Upsilon$ interprets $\Sigma$ if for all specs in $\Sigma$, one of the following must be true:
1. If the spec is an open tycon \((\rho, n)\), then \(\Upsilon(\rho) = \tau^n\) or \(\Upsilon(\rho) = \mathbb{C}^\lambda\) such that \(\Upsilon \vdash \mathbb{C}^\lambda :: \Omega^n \to \Omega\).

2. If the spec is a structure \((\rho, \Sigma)\), then \(\Upsilon(\rho) = R'\).

3. If the spec is a functor \((\rho, \Sigma^f)\), then \(\Upsilon(\rho) = \psi\).

Lemma 34

If \(\Upsilon\) interprets the extracted signature of \(\Gamma\) and \(\Gamma, \Upsilon \vdash \text{str}\) \(=((\Sigma, R), \varphi)\), then \(R\) interprets \(\Sigma\).

Proof: The proof for module declaration elaboration is by inspection. For structure expressions, the base structure, and let expression rely on the proof for module declaration. The functor application and two ascription cases rely on signature matching preserving the \text{interprets} relationship. For signature matching, we can see that the resultant entity environment does indeed interpret the spec signature by inspection.

C.3.3 Module Declaration Elaboration

Lemma 35 (Synchronization of Static and Entity Environments)

If \(\Gamma, \Upsilon \vdash \Gamma \Rightarrow decl\) \((\eta, \Gamma', \Upsilon')\) and \(EV(\Gamma) \subseteq dom(\Upsilon)\), then \(EV(\Gamma') \subseteq dom(\Upsilon')\).

Proof: [by induction on the derivations]

Rules 7.26, 7.27, and 7.28 \(\mathcal{X}, \mathbb{C}^\lambda, \tau^n\) are semantic and therefore contain no entity variables.

Rule 7.29 By lem. 32, \(EV(M) \subseteq dom(\Upsilon) \cup dom(R)\). By induction, \(EV(\Gamma') \subseteq dom(\Upsilon')\). It follows that \(EV([X \mapsto (\rho, M)]\Gamma') \subseteq dom([\rho \mapsto R]\Upsilon')\).

Rule 7.30 By lem. 39, \(EV(\Gamma[X \mapsto (\rho_x(\Sigma_x, R_x)])) \subseteq dom(\Upsilon[\rho_x \mapsto R_x])\). By lem. 32, \(EV(\Sigma_{res}, R_{res}) \subseteq dom(\Upsilon) \cup dom(R_{res})\).
The following lemmas guarantee that semantic signatures and semantic functor signatures in full signatures and full functor signatures are closed by the structure entity and functor entity respectively.

**Lemma 36**

If \( \Gamma, \Upsilon \vdash \text{structure } X = \text{strex}_p, d^m \Rightarrow_{\text{decl}} (\cdot, [X \mapsto (\rho, (\Sigma, R))] \Gamma', \cdot) \), then for all relativized paths \( \vec{\rho} \) in \( \Sigma \), there exists an entity \( \upsilon \) such that \( R(\vec{\rho}) = \upsilon \).

**Lemma 37**

If \( \Gamma, \Upsilon \vdash \text{functor } f(X : \text{sig}_p) = \text{strex}_p, d^m \Rightarrow_{\text{decl}} (\cdot, [f \mapsto (\rho, (\Pi \rho_x : \Sigma_x, \Sigma_{res}, \psi))] \Gamma'', \cdot) \) and \( \Upsilon \vdash \Sigma_x \uparrow \Upsilon_x \), then for all entity paths \( \vec{\rho} \) in \( \Sigma_{res} \), \( (\Upsilon_x, \Upsilon)(\vec{\rho}) \) is defined.

### C.3.4 Signature Instantiation

**Lemma 38 (Signature Instantiation Terminates)**

\( \Upsilon \vdash \Sigma \uparrow \Upsilon \) terminates.

**Proof:** [by induction on length of semantic signature] All the rules work on a strict sub-sequence of the semantic signature \( \Sigma \) except rule 7.20 (the structure spec instantiation rule). Even in that case, the structure spec signature \( \Sigma' \) is clearly less than the length of \( [x \mapsto (\rho, \Sigma')] \Sigma \).

**Lemma 39**

If \( \Upsilon^{\text{clo}}, \Upsilon^{\text{lcl}} \vdash \Sigma \uparrow \Upsilon \), then \( \text{EV}(\Sigma) \subseteq \text{dom}(\Upsilon) \cup \text{dom}(\Upsilon^{\text{clo}}) \cup \text{dom}(\Upsilon^{\text{lcl}}) \)

**Signature Matching**

**Lemma 40**

If \( \Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{\text{match}} (R_c, \varphi_c) \), then for all \( [x \mapsto s] \in \Sigma_s, [x \mapsto s'] \in \Sigma_a \) such that \( R_c(s) = s' \).
Proof: [by straightforward induction on the derivation] ■

The induction on the two lemmas, lem. 41 and 42 require that they be proved simultaneously.

Lemma 41
If \( \Upsilon, \Sigma_a, \rho_u \vdash \Sigma_s \Rightarrow_{\text{coerce}} (\Upsilon', \eta) \), \( EV(\Sigma_s) \subseteq \text{dom}(\Upsilon) \), \( R_u \subseteq \Upsilon \), and \( \forall \bar{\rho} \in \text{dom}(R_u).R_u(\bar{\rho}) = \Upsilon(\bar{\rho}) \), then \( \Upsilon[\rho_u \mapsto R_u] \vdash \eta \downarrow_{\text{decl}} \Upsilon' \).

Proof: [by simultaneous induction on the derivation of \( \Rightarrow_{\text{coerce}} \) and the lemma below]

Rule 7.46 \( \Upsilon[\rho_u \mapsto R_u] \vdash [\rho =_{\text{def}} \rho_u \rho_a] \downarrow_{\text{decl}} [\rho \mapsto v] \) such that \( v = \Upsilon[\rho_u \mapsto R_u](\rho_u \rho_a) = R_u(\rho_a) = \Upsilon(\rho_a) \). By induction, we are done.

Rule 7.50 \( \Upsilon[\rho_u \mapsto R_u] \vdash \rho_u \rho_x \downarrow_{\text{str}} \Upsilon[\rho_u \mapsto R_u](\rho_u \rho_x) = \Upsilon(\rho_x) \). \( \Upsilon[\rho_u \mapsto R_u] \vdash [\rho_s =_{\text{str}} \varphi_c] \downarrow_{\text{decl}} [\rho_s \mapsto v] \) such that \( v = R_c \) because \( \Upsilon \vdash \varphi_c \downarrow_{\text{str}} R_c \) by lem. 42.

All the other cases are essentially the same as the above two cases. ■

Lemma 42
If \( \Upsilon \vdash ((\Sigma_a, R_a), \varphi) : \Sigma_s \Rightarrow_{\text{match}} (R_c, \varphi_c) \), \( \Upsilon \vdash \varphi \downarrow_{\text{str}} R_a \), and \( EV(\Sigma_a) \subseteq \text{dom}(R_a) \), then \( \Upsilon \vdash \varphi_c \downarrow_{\text{str}} R_c \).

Proof: This lemma follows from lem. 41. ■

C.3.5 Translation

Lemma 43 (Correctness of Coercion)
If \( E \vdash \{\text{lts}_u\} :: k \), \( E \vdash \rho_u : \{\text{lts}_u\} \), \( E \vdash \alpha :: k' \), \( E \vdash c :: k' \), for all \( l \in \text{dom}(\text{lts}) l \in \text{dom}(\text{lts}_u) \) and \( t\{c/\alpha\} = t_u \) such that \( l : t \in \text{lts} \) and \( l : t_u \in \text{lts}_u \), and \( \text{coerce}(\rho_u, c, \text{lts}) = e_\omega \), then \( E \vdash e_\omega : \{\text{lts}\{c/\alpha\}\} \).
Proof: [by induction on the structure of lts] If \( lts = \{ \} \), then \( e_\omega = \{ \} \). \( E \vdash \{ \} : \{ \} \) by rule 8.14. Otherwise, \( lts = \{ l_0 : t_0 \} \cup lts' \) in which case \( \text{coerce}(\rho_u, c, \{ l_0 : t_0 \} \cup lts') = \{ l_0 = \rho_u \cdot l_0 \} \cup les \) such that \( les = \text{coerce}(\rho_u, c, lts') \). By induction, \( E \vdash \text{coerce}(\rho_u, c, lts') : \{ lts'\{c/\alpha\}\} \). By assumption, \( l_0 \in \text{dom}(lts_u) \). Therefore, \( l_0 : t_{u_0} \in lts_u \). By assumption, \( t_{c_0}\{c/\alpha\} = t_{u_0} \) where \( l_0 : t_{c_0} \in lts_c \). Certainly, \( \{ l_0 = \rho_u \cdot l_0 \} : t_{u_0} = t_{c_0}\{c/\alpha\} \). It follows that \( E \vdash e_\omega : \{ l_0 : t_{c_0}, lts'\}\{c/\alpha\} = \{ lts\{c/\alpha\}\} \).

Lemma 44
For all \( \Sigma \), there exists a \( k \) such that \( \vdash \Sigma \rightsquigarrow_{\text{knd}} k \).

Proof: [by induction on the structure of \( \Sigma \)]

Lemma 45
For all \( C^n_f \), there exists a \( t \) such that \( \vdash C^n_f \rightsquigarrow_{n_f}^{tyc} t \).

Proof: [by induction on the structure of \( C^n_f \)]

Lemma 46 (Closed semantic types translate)
1. If \( EV(T) \subseteq \text{dom}(\Upsilon) \), then \( \Upsilon \vdash T \rightsquigarrow_{t}^{tyc} t \)
2. If \( EV(\Sigma) \subseteq \text{dom}(\Upsilon) \), then \( \Upsilon \vdash \Sigma \rightsquigarrow_{t}^{tyc} t \).
3. If \( EV(C^s) \subseteq \text{dom}(\Upsilon) \) and \( TV(C^s) \subseteq \Delta \), then \( \Delta, \Upsilon \vdash C^s \rightsquigarrow_{t}^{tyc} t \).

Proof: [by induction on the size of the term]

1. The proof for this derivation follows directly from the proof of (3)
2. We consider the cases.

\( \Sigma = \emptyset_{\text{sig}} \) By rule 8.52
\( \Sigma = \text{spec}\Sigma' \) Consider the cases for spec.

153
By assumption $EV(T) \subseteq \text{dom}(\Upsilon)$. By induction (1), $\Upsilon \vdash T \leadsto^t \text{type } t$. By induction (2), $\Upsilon \vdash \Sigma' \leadsto^t \text{type } t'$. By rule 8.53, $\Upsilon \vdash [x \mapsto T] \Sigma' \leadsto^t \{x : t\} \psi t'$.

By induction (2), $\Upsilon \vdash \Sigma' \leadsto^t \text{type } t'$. By rule 8.54, we are done.

$(\rho, n)$ Similar reasoning as above

$(\rho, (\Sigma_0, R))$ By induction (2), $\Upsilon \vdash \Sigma_0 \leadsto^t \text{type } t$ and $\Upsilon \vdash \Sigma' \leadsto^t \text{type } t'$. By rule 8.56, we are done.

$(\rho, (\Pi \rho_x : \Sigma_x, \Sigma, \psi))$ By induction (2), $\Upsilon \vdash \Sigma_x \leadsto^t \text{type } t_x$, $\Upsilon \vdash \Sigma \leadsto^t \text{type } t$, and $\Upsilon \vdash \Sigma' \leadsto^t \text{type } t'$. By rule 8.57, we are done.

3. Rule 8.48 is by assumption. Rule 8.49 relies on normalization of semantic tycon evaluation and lem. 45.

Lemma 47 (Tycon Substitution)

If $E \vdash e_\omega \leadsto^t \text{type } t$, $E(\alpha) = k$, and $E \vdash c :: k$, then $E \vdash e_\omega\{c/\alpha\} \leadsto^t \text{type } t\{c/\alpha\}$.

Proof: [by induction on the derivation of $\leadsto^t \text{type}$] ■

Lemma 48 (Correctness of Type Synthesis)

If $R$ is closed, $R \vdash \Sigma \leadsto^t \text{type } \text{lts}$, $l : T \in \Sigma$, $R \vdash T \leadsto^t \text{type } t$, and $l : t' \in \text{lts}$ such that $t'$ is not a record type, then $t = t'$.

Lemma 49 (Correctness of Relativized Type Translation)

If $R(T) = R'(T')$ and $R \vdash T \leadsto^t \text{type } t$, then $R' \vdash T' \leadsto^t \text{type } t$.

Proof: ■

Lemma 50

If $R$ interprets $\Sigma, \vdash \Sigma \leadsto^\text{knd } k$, and $R \vdash \Sigma \leadsto^\text{spec } \text{lcs}$, then $R \vdash \{\text{lcs}\} :: k$. 

154
Proof: [by induction on the derivation of $R \vdash \Sigma \sim_{spec} lcs$] Both $\vdash \Sigma \sim_{knd} k$ and $R \vdash \Sigma \sim_{spec} lcs$ ignore $[x \mapsto \mathbb{T}]$ and $[x \mapsto \mathbb{C}^\lambda]$ specs. For open tycon specs, $[x \mapsto (\rho, n)] \Sigma' \sim_{knd} \{\rho : \Omega^n \rightarrow \Omega\} \uplus k'$ by rule 8.35. By rule 8.42, $\text{lcs} \rho = \text{inj}(R(\rho)), \text{lcs}'$ such that $R \vdash \Sigma' \sim_{spec} lcs'$ for $\Sigma = [x \mapsto (\rho, n)] \Sigma'$. By def. 4, $R(\rho) = \tau^n$. Therefore $\text{inj}(R(\rho)) :: \Omega^n \rightarrow \Omega$. ■

Lemma 51
If $R$ interprets $\Sigma, \vdash \Sigma \sim_{knd} k, (\Sigma, R) \sim_{tyc} c$, then $R \vdash c :: k$.

Proof: This proof follows directly from lem. 50. ■

Theorem 2 (Translation Preserves Well-Typing)
If $EV(\Gamma) \subseteq \text{dom}(\Upsilon), \Gamma, \Upsilon \vdash \text{strexp} \Rightarrow_{str} (\text{strexp}, (\Sigma, R), \_)$, $\Upsilon \vdash \Gamma \leftarrow \Sigma'$, $\Upsilon \vdash \Sigma' \sim_{type} \{\text{lts}\}$, and $\text{strexp} \sim_{exp} \text{e}_\omega$, then $\text{env}(\text{lts}) \vdash \text{e}_\omega : t$ and $R \vdash \Sigma \sim_{type} t$.

If $EV(\Gamma) \subseteq \text{dom}((\Upsilon), \Gamma, \Upsilon \vdash \text{d}^m \Rightarrow_{decl} (\text{d}^m, \_), \Gamma', \Upsilon')$, $\Upsilon \Upsilon' \vdash \Gamma' \leftarrow \Sigma'$, $\text{env}(\text{lts}) \vdash \text{d} ::_{dec} \text{E}'$ where $\Upsilon \vdash \Sigma \sim_{type} \{\text{lts}\}$, $\Upsilon \Upsilon' \vdash \Sigma' \sim_{type} \{\text{lts}'\}$, and $\text{env}(\text{lts}') = \text{E}'$.

Proof: [By induction on the derivation of $\Gamma, \Upsilon \vdash \text{strexp} \Rightarrow_{str} (\text{strexp}, (\Sigma, R), \varphi)$]

Structure expressions

Rule 7.31 (structure path) The rule says $\Gamma, \Upsilon \vdash p \Rightarrow_{str} (p(\langle \rho \rangle), (\Sigma, R), \varrho)$. By inversion of $\Rightarrow_{str}$, $\Gamma(p) = (\varrho, M)$ where $M = (\Sigma, R)$. By definition, $[x \mapsto (\rho, M)] \in \Gamma$ such $p$ ends in a singleton $x$. By definition of signature extraction (rule 7.41), $[x \mapsto (\rho, \Sigma)] \in \Sigma'$.

By rule 8.27, $p(\langle \rho \rangle) \sim_{exp} \varrho$. By lem. 36, $EV(\Sigma) \subseteq \text{dom}(R)$. By lem. 46, $R \vdash \Sigma \sim_{type} t$ and $\Upsilon \vdash \Sigma' \sim_{type} t'$. By rule 8.56, $\Upsilon \vdash [x \mapsto (\rho, M)]\Sigma' \sim_{type} \{\rho : t\} \uplus t' = \{\text{lts}\}$. By rule 8.15, $\text{env}(\text{lts}) \vdash \varrho : t$.

Rule 7.32 (base structure) By induction and reasoning similar to the above.
Rule 7.33 (application) The nontrivial case is the application rule.

\[ \text{str}^p = p(\langle \bar{\rho}, \rho_{\text{par}} \rangle)(\text{str}^p_x((M_c, \rho_u))). \]

By inversion of rule 7.33, \( \Gamma(p) = (\bar{\rho}, (\Pi_{\text{par}} : \Sigma_{\text{par}}.\Sigma_{\text{body}}, \psi)) \) (b.1), \( \Gamma, \Upsilon \vdash \text{str}^p_x \Rightarrow \text{str} (M_x, \varphi_x) \) (b.2), \( \Upsilon \vdash (M_x, \varphi_x) : \Sigma_{\text{par}} \Rightarrow \text{match} (R_c, \varphi_c) \) (b.3), \( \varphi_{\text{app}} = \bar{\rho}(\varphi_c) \) (b.4), and \( \Upsilon \vdash \varphi_{\text{app}} \downarrow_{\text{str}} R_{\text{app}} \) (b.5).

This is a caveat for (b.1). \( \Gamma(p) \) contains a \( \psi \) if \( p \) is a real functor. If it is a formal functor, \( \Gamma(p) \) will only have the entity variable and functor signature. The \( \psi \) will have to be produced from \( \Upsilon(\bar{\rho}) \). In either case, we can get a full functor signature.

By assumption, \( \Upsilon \vdash \Gamma \leftarrow \Sigma' \) and \( \Upsilon \vdash \Sigma' \sim_{\text{type}} \{\text{lts}\} \). Let \( \rho \) be the last entity variable in \( \bar{\rho} \) and \( f \) be the final symbol in \( p \). Then by definition of signature extraction, \( [f \mapsto (\rho, \Pi_{\text{par}} : \Sigma_{\text{par}}.\Sigma_{\text{body}})] \in \Sigma' \). Thus, \( \Upsilon \vdash [f \mapsto (\rho, \Pi_{\text{par}} : \Sigma_{\text{par}}.\Sigma_{\text{body}})]\Sigma'' \sim_{\text{type}} \{\rho : \forall_{\text{par}} :: k.\text{t}_{\text{par}} \rightarrow t_{\text{body}}\} \equiv t'' \) By inversion, \( \vdash \Sigma_{\text{par}} \sim_{\text{knl}} k \) (c.1), \( \Upsilon \vdash \Sigma_{\text{par}} \sim_{\text{type}} t_{\text{par}} \) (c.2), \( \Upsilon \vdash \Sigma_{\text{body}} \sim_{\text{type}} t_{\text{body}} \) (c.3), and \( \Upsilon \vdash \Sigma'' \sim_{\text{type}} t'' \) (c.4).

Because \( \{\rho : \forall_{\text{par}} :: k.\text{t}_{\text{par}} \rightarrow t_{\text{body}}\} \in \text{lts} \), \( E(\bar{\rho}) = \forall_{\text{par}} :: k.\text{t}_{\text{par}} \rightarrow t_{\text{body}} \).

By assumption and rule 8.29, \( \Gamma, \Upsilon \vdash p(\langle \bar{\rho}, \rho_{\text{par}} \rangle)(\text{str}^p_x((M_c, \rho_u))) \sim_{\text{exp}} \) let \( \rho_u = e_{\omega_x} \) in \( \bar{\rho}[c](e_{\omega_c}) \) where \( \vdash M_c \sim_{\text{tyc}} c, \text{str}^p_x \sim_{\text{exp}} e_{\omega_x} \) (a.0), \( R_c \vdash \Sigma_{\text{par}} \sim_{\text{type}} \{\text{lts}_c\} \) (a.1), and \( e_{\omega_c} = \text{coerce}(\rho_u, c) \) (a.2). Let \( M_c = (\Sigma_{\text{par}}, R_c) \) and \( t_{\text{par}} = \{\text{lts}_c\} \).

By rule 8.9, \( E \vdash \bar{\rho} : \forall_{\text{par}} :: k.\text{t}_{\text{par}} \rightarrow t_{\text{body}} \). By lem. 51, \( E \vdash c :: k \). By rule 8.13, \( E \vdash \bar{\rho}[c] : \text{t}_{\text{par}}\{c/\rho_{\text{par}}\}_{\text{tyc}} \rightarrow t_{\text{body}}\{c/\rho_{\text{par}}\}_{\text{tyc}} \).

Let \( M_x = (\Sigma_x, R_x) \). By (b.2), (a.0), and the original assumptions, \( E \vdash e_{\omega_x} : t_x \) and \( R_x \vdash \Sigma_x \sim_{\text{type}} \{\text{lts}_x\} \).

By definition, \( \text{coerce}(\rho_u, \{\text{lts}_par\}) = \{x_1 = \rho_u.x_1, \ldots, x_n = \rho_u.x_n\} \) where \( x_i : t_i \in \text{lts}_par \).

\( E[\rho_u = e_{\omega_x}] \vdash x_1 = \rho_u.x_1 : \text{dec} x_1 : t'_1 \) such that \( \forall i \in [1, n] x_i : t'_i \in \text{lts}_x \).

By definition of tycon synthesis, \( c = \{\text{lcs}\} \) where \( R_c \vdash \Sigma_{\text{par}} \sim_{\text{spec}} \text{lcs} \) such that:
1. \( \forall [x \mapsto (\rho_i, n)] \in \Sigma_{\text{par}} \rho_i = c_i \in \text{lcs} \) such that \( c_i = \text{inj}(R_c(\rho_i)) \). By lem. 40, for all \( [x \mapsto (\rho_i, n)] \in \Sigma_{\text{par}}, [x \mapsto s'] \in \Sigma_x \) such that \( R_c(\rho_i) = R_x(s') \). Therefore, \( t_i\{\rho_i = \text{inj}(R_c(\rho_i))/\rho\}\text{tyc} = t_i\{\rho_i = R_x(s')/\rho\}\text{tyc} = t'_i \).

2. \( \forall [x \mapsto (\rho_i, \Sigma)] \in \Sigma_{\text{par}} \rho_i = \{\text{lcs}_i\} \) such that \( R(\rho_i) \vdash \Sigma \rightsquigarrow \text{spec } \text{lcs}_i \). This case goes through by induction.

3. \( \forall [x \mapsto (\rho_i, \Pi'_{\text{par}} : \Sigma'_{\text{par}}, \Sigma'_{\text{body}})] \in \Sigma_{\text{par}} \rho_i = \lambda_{\text{par}}^i : k.c' \) such that \( \vdash \Sigma'_{\text{par}} \rightsquigarrow \text{knd } k \) and \( \vdash R_c(\rho_i) \rightsquigarrow_{\text{tyc}} \text{stexp } c' \).

(d.1)

Thus, \( E[\rho_u = e_{\omega x}] \vdash \text{coerce}(\rho_u, \{\text{lts}_{\text{par}}\}) : \{\text{lts}_{\text{par}}\}\{c/\rho\}\text{tyc} \). Recall that \( e_{\omega c} = \text{coerce}(\rho_u, \{\text{lts}_{\text{par}}\}) \). Let \( E' = E[\rho_u = e_{\omega x}] \). By rule 8.11, \( E' \vdash \bar{\rho}[c](e_{\omega c}) : t_{\text{body}}\{c/\rho_{\text{par}}\} \).

Let \( t_{\text{body}} = \{\text{lts}_{\text{body}}\} \). Since \( \Upsilon \subseteq R_{\text{app}} \), by (c.3) and reasoning similar to (d.1), \( R_{\text{app}} \vdash \Sigma_{\text{body}} \rightsquigarrow \text{type } \text{lts}_{\text{body}}\{c/\rho_{\text{par}}\} \).

**Module declarations** These cases are straightforward given the structure expression proof.
## APPENDIX D

### NOTATION INDEX

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p,x,K,\Omega,C^s,C^\lambda,T,\overline{C^s}$ typ</td>
<td>surface syntax</td>
<td>49</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>kind environment</td>
<td>48</td>
</tr>
<tr>
<td>$C^s_i$</td>
<td>indexing a vector of monotypes</td>
<td>47</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>arity</td>
<td>51</td>
</tr>
<tr>
<td>$\Upsilon[\rho \mapsto v]$</td>
<td>entity environment</td>
<td>59</td>
</tr>
<tr>
<td>$\langle \Upsilon^{lc}, \Upsilon^{do} \rangle$</td>
<td>structure entity</td>
<td>59</td>
</tr>
<tr>
<td>$\langle \lambda \rho.\varphi; \Upsilon \rangle$</td>
<td>functor entity</td>
<td>59</td>
</tr>
<tr>
<td>$\llbracket \rrbracket$</td>
<td>base entity expression</td>
<td>59</td>
</tr>
<tr>
<td>$EV(\cdot)$</td>
<td>free entity variables</td>
<td>62</td>
</tr>
<tr>
<td>$\Upsilon(\vec{\rho})$</td>
<td>entity environment lookup</td>
<td>62</td>
</tr>
<tr>
<td>$\Upsilon \Upsilon'$</td>
<td>entity environment concatenation</td>
<td>62</td>
</tr>
<tr>
<td>$R(\vec{\rho})$</td>
<td>structure entity lookup</td>
<td>62</td>
</tr>
<tr>
<td>$\Upsilon \subseteq \Upsilon'$</td>
<td>entity environment containment</td>
<td>62</td>
</tr>
<tr>
<td>$\mathbf{EP}, \Sigma_{ep}(p), \Sigma(p), \Gamma_{ep}(p), \Gamma(p)$</td>
<td>entity path construction</td>
<td>78</td>
</tr>
</tbody>
</table>
REFERENCES


