Program Modules, Separate Compilation, and Intermodule Optimisation

Ph.D. Thesis

Martin Elsman

Department of Computer Science
University of Copenhagen

Accepted by
the Faculty of Science
University of Copenhagen

January 1999

Supervisor
Mads Tofte, University of Copenhagen

Thesis Committee
Xavier Leroy, INRIA Rocquencourt
Greg Morrisett, Cornell University
Hanne Riis Nielson, Aahus University (Chair)

Abstract

This thesis is about a framework for elaborating and interpreting module language constructs at compile time in such a way that (1) arbitrary compile time information about declared identifiers of a module may be propagated to other modules and (2) no code is generated for module language constructs. The framework for interpreting module language constructs is called static interpretation. More information about referenced identifiers than can be obtained from programmer provided interfaces is necessary for analyses such as region inference. Further, many other analyses improve significantly from availability of analysis specific information about referenced identifiers. Static interpretation facilitates intermodule optimisation, yet, it still supports a variant of separate compilation called cut-off incremental recompilation.

The thesis is divided into three parts. The first part is about the static properties of a small language called ModML, which features a small Core language and the essential constructs of the Standard ML Modules language. The second part develops the framework for static interpretation by showing how ModML can be compiled into an explicitly typed intermediate language. The third part describes the ML Kit with Regions compiler, which is based on the techniques developed in part one and part two.

Contents

Preface

This document is the revised version of the thesis. The thesis was submitted July 20th, 1998 in partial fulfilment of the requirements for the degree of Doctor of Philosophy, and recommended for acceptance by the thesis committee December 15th, 1998. The thesis is accepted by the Faculty of Science, University of Copenhagen.

Acknowledgments

Foremost, I would like to thank my advisor Mads Tofte for giving me inspiration and excellent supervision; much of the work put in the thesis is the result of his teaching and guidance.

Thanks to the other members of the ML Kit team, that is, Peter Bertelsen, Lars Birkedal, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Mads Tofte, for making work a fun thing to do.

Also thanks to the people in the TOPPS group at DIKU for numerous fine discussions and for a great research atmosphere.

Special thanks go to Úlfar Erlingsson, Greg Morrisett, and the rest of the people at Cornell University for making my stay there enjoyable and educating.

I would also like to thank the thesis committee for providing comments for the revised version of the thesis.

Finally, I would like to thank my family and my wife Sissel for their love and support.

Martin Elsman
Berkeley
January 1999

Chapter 1

Introduction

Modularity arose from the need to divide programs into program units for separate compilation. Since then, the concept of modularity has become a key element of modern software engineering [Car97]. The concept of modularity now also covers name space management and abstraction mechanisms such as parametric modules and abstract types. As software projects get larger, programming language support for modularity becomes more and more important for software development, for software maintenance, and for software reuse.

On the other hand, only rarely does modularity come for free; when a program unit is compiled, many programming environments impose a tradeoff between what information for referenced identifiers is available to the compiler and what information is hidden for the programmer. Some information may be useful to the programmer (e.g., simple type information,) other information may not (e.g., in what machine registers does a function expect its arguments.) One reason for this tradeoff is that support for separate compilation has been highly integrated in source languages.

In this thesis, we develop a general framework for separate compilation that exists independently of the source language of the compiler and thus does not impose any such tradeoff; we call the framework *cut-off incremental recompilation* because program units managed by the framework are compiled incrementally (i.e., a program unit may be compiled only when the program units on which it depends have been compiled) and because a program unit need not necessarily be recompiled if another program unit, on which it depends, is modified. The framework is based both on properties of the source language and on each translation step in the compiler. An

important property of the framework is that it may coexist with a module language; thus, the framework does not compromise software engineering principles.

1.1 Modularity and Abstraction

Standard ML [MTHM97] is a statically typed language that supports a Core language for programming in the small and a Modules language for programming in the large. The Standard ML Modules language [MTHM97] is a well-studied language that builds upon the notions of structure, signature, and functor. Briefly, a *structure* is a sequence of declarations that may include declarations of values, of types, and even of other structures. A *signature* specifies components of a structure and may thus be used to describe a structure without actually supplying an implementation. A *functor*, then, is a function mapping structures to structures.

The type system of Standard ML has a mechanism for generating a fresh type (a name), which is then considered distinct from all other previously declared types. This mechanism is used to support abstract datatypes and what is called opaque signature constraints, which restrict the view of a structure to that expressed by a signature. The name based static semantics of Standard ML is difficult to reason about mainly because generated names cannot be renamed once they are generated. In the first part of this thesis, we present a static semantics for a subset of the language Standard ML. We call the language ModML; like Standard ML, ModML is divided into a Core language and a Modules language. The Modules language has most of the essential features of Standard ML Modules whereas the Core language is very small; it is there to account for the interaction between the Modules language and the Core language.

The static semantics of ModML uses type abstraction, instead of a mechanism for generating fresh names, to model abstract datatypes and opaque signature constraints. Type abstraction is also used in the module systems by Harper and Lillibridge [HL94] and by Leroy [Ler94, Ler95]. Moreover, Leroy demonstrates [Ler96] that a name based module system is equivalent to a module system based on type abstraction. The type systems of these languages differ from ModML, however, in that ModML is founded on mathematical objects that are defined independently of the syntax of ModML. The mathematical objects on which ModML is founded are much the same as

the semantic objects on which Standard ML is founded. Independently from the work we have done here, Russo [Rus98] has developed a static semantics for a subset of Standard ML Modules, which is much similar to the static semantics for ModML that we give here. In particular, Russo also formulates type generativity by type abstraction. Moreover, he demonstrates that his static semantics for Modules accepts the same set of language phrases as a name based static semantics. Although Russo extends the features of Standard ML Modules to support higher-order [MT94, Ler95, HL94] and first-class modules [MP88, Jon95], he does not demonstrate type soundness for his language.

An important property of Standard ML is type soundness; that is, well-typed Standard ML programs do not go wrong. We demonstrate this property for the language ModML via a type preserving interpretation of ModML programs into an intermediate language called IntML for which a type soundness result exists. The result is proof that the type abstraction mechanism of ModML is sound. The interpretation of ModML into IntML is called *static interpretation*, because all Modules language phrases are eliminated in the process and because the interpretation does not depend on actual runtime values.

1.2 Separate Compilation

In general, a separate compilation framework allows the programmer to divide a program into several program units, which can then be compiled in isolation. There are at least three aspects to the importance of separate compilation. First, there is the modularity aspect as discussed in the preceding section; a separate compilation framework provides the programmer with a simple mechanism for organising programs into several program units. Then, there is the compiler aspect; machines are often too small and compilers are often not efficient enough to handle large bodies of code at once. Third, when a software project has been compiled once, recompilation upon modification of one or more program units must be fast.

Different separate compilation frameworks have different characteristics with respect to what program units are recompiled upon modification of a program unit. The simplest framework will recompile all program units; this method is called *big bang* [ATW94]. Conventional recompilation and smart recompilation, as presented below, assume that the programmer provides

an *interface* for each program unit in the program; such interfaces provide specifications (e.g., types) for each declared identifier of the corresponding program unit. The conventional framework for separate compilation, as implemented by the UNIX program `make`, has the following characteristics:

> *A program unit must be recompiled whenever (1) its own implementation changes, or (2) an interface changes upon which the program unit depends.*

Modifying a comment or adding extra specifications to an interface cause unnecessary recompilation. A more relaxed version of the preceding method is called *smart recompilation* [Tic86]:

> *A program unit must be recompiled whenever (1) its own implementation changes, or (2) if it references an identifier whose specification has changed.*

Frameworks for separate compilation that allow a program unit to be compiled based on interfaces, provided by the programmer, for those program units on which the program unit depends, support cut-off (or true) separate compilation; that is, such frameworks allow a program unit to be compiled before the program units on which it depends have been compiled.

Shao and Appel describe in [SA93] how *smartest recompilation* may be implemented:

> *A program unit never needs to be recompiled unless its own implementation changes.*

Smartest recompilation employs type inference to infer types for undeclared identifiers of a program unit. Because some type inference is deferred till link time, error reporting for some conflicts between declarations and uses may also be delayed; such delays are in conflict with the rule that programmers generally prefer to receive error messages as early as possible. Consult [ATW94] for a thorough discussion of several variations of the preceding recompilation schemes.

1.3 Implementation Transparency

Cut-off incremental recompilation allows for arbitrary compile time information about declared identifiers to propagate across program unit boundaries.

The framework does not build on interfaces and thus lacks cut-off separate compilation; it has the following characteristics:

> *A program unit must be recompiled if either (1) the program unit has changed or (2) information about used identifiers of the program unit has changed.*

Moreover, the framework supports what we call *implementation transparency*; that is, arbitrary information about a declared identifier may be propagated across program unit boundaries. Such information may include intermediate representations of in-line code and information about calling conventions for function identifiers.

In Section 1.1, we described how static interpretation interprets the language ModML into an intermediate language called IntML. The language IntML is much simpler than ModML; in IntML programs, all types have become known and all Modules language constructs have been eliminated by flattening of structures and by specialisation of functors. This property is nice, for at least two reasons. First, programmers do not always want to compromise performance; without this property, programmers are sometimes encouraged to in-line declarations from other modules manually, thereby violating software engineering principles. Second, and most importantly, the specialisation of functors allows arbitrary information about declared identifiers to propagate across module boundaries at compile time; the specialisation of functors was initiated by the same reason that the separate compilation framework does not support cut-off separate compilation. Cut-off incremental recompilation addresses the problem traditionally associated with functor specialisation: that if a program is heavily functorised, then all code generation is delayed till link time. With cut-off incremental recompilation, only the functor applications for which the functor or compiler assumptions have changed are recompiled—and if there is only one application of each of those functors, this recompilation involves no more work than recompiling each of the functor bodies once.

The specialisation of ModML functors is much similar to how Ada generic packages and C++ templates are usually compiled. Besides from working well with cut-off separate compilation, ModML functors can be fully type checked at declaration time—as opposed to application time.

1.4 Outline

There are three parts. The first part is about the static properties of the language ModML. In Chapter 2, we define the grammar for ModML and we present the static semantics for the language. In Chapter 3, we study some simple properties of ModML, many of which we use to demonstrate other properties of the language in succeeding chapters. In Chapter 4, we demonstrate that elaboration of a ModML program phrase depends only on assumptions for those identifiers that occur free in the program phrase; this property is important for separate compilation. In Chapter 5, we show that it is possible to eliminate opaque signature constraints from a ModML program by translating them into transparent ones such that if the original program elaborates then so does the translated program.

The second part is about compilation of program components. In Chapter 6, we present a framework for managing separate compilation, which we call cut-off incremental recompilation. The framework allows for arbitrary compile time information about declared identifiers being propagated across program unit boundaries to those program units that use the declared identifiers. In Chapter 7, we present an explicitly typed language called IntML, for which we demonstrate type soundness; that is, we show that, using operational semantics, well-typed IntML programs do not go wrong when evaluated. In Chapter 8, we present an interpretation of ModML programs into the language IntML. Further, we demonstrate that all well-typed ModML programs may be interpreted into a well-typed IntML program.

The third part is about the ML Kit with Regions (or just the Kit). In Chapter 9, we describe the overall structure of the Kit. We also show how the techniques in part one and part two are used in the Kit to support separate compilation and Standard ML Modules. In Chapter 10, we give an overview of the back-end phases of the Kit and describe how these phases fit into the framework for separate compilation that we presented in Chapter 6 and Chapter 8.

Finally, we give a conclusion in Chapter 11.

Part I

A Module Language

Chapter 2

The Language ModML

In this chapter, we present the language ModML. The language ModML features a Core functional language for programming in the small and a Modules language for programming in the large. ModML is essentially a small subset of the programming language Standard ML (SML'97). However, ModML is small because we want to study properties of the language with mathematical rigour. Still, the Core language of ModML has support for polymorphic values, higher-order functions, and generative datatypes, and the Modules language has support for signatures, structures, and functors. To simplify the presentation, ModML does not support signature declarations.

The emphasis here is on the static properties of ModML. The static semantics for ModML describes what programs should be accepted by a compiler and what programs should not; the process of determining what programs are acceptable and what programs are not is called *elaboration*. Elaboration relates syntactic constructs to mathematical objects based on some background that relates free identifiers of the syntactic construct to mathematical objects. The major difference between the static semantics of ModML and that of SML'97 is that ModML uses type abstraction to model type generativity. This modification makes it easier to demonstrate important properties of the language.

We do not give a dynamic semantics for ModML, directly. Instead, in Chapter 8, we present an interpretation of ModML programs into a language called IntML, for which a dynamic semantics is given in Chapter 7. There is a type soundness result for the language IntML, thus, type soundness for ModML can be established, indirectly.

In the section to follow, we introduce some syntactic notation. Then,

in Sections 2.2 through 2.4, we present a language of type expressions and associated rules for elaboration. In Sections 2.5 through 2.7, we present the Core language of ModML. Further, in Sections 2.8 through 2.12, we present the grammar for expressing signatures together with rules for elaborating signature constructs. Finally, in Sections 2.14 through 2.17, we present the grammar and elaboration rules for writing modules in ModML.

2.1 Identifiers and Syntactic Notation

We divide *identifiers* into classes VId of *value identifiers*, TyCon of *type constructors*, TyVar of *type variables*, StrId of *structure identifiers* and FunId of *functor identifiers*. We use *vid*, *tycon*, *strid* and *funid* to range over value identifiers, type constructors, structure identifiers and functor identifiers, respectively. Further, we use *tyvar* and α to range over type variables.

For each class X of identifiers, ranged over by x, there is a class Long$X$, ranged over by *longx*, defined as follows:

longx	::=	x	identifier
	|	$strid_1.\cdots.strid_n.x$	qualified identifier, $n \geq 1$

Similarly, there is a class XSeq, ranged over by *xseq*, describing sequences of identifiers, defined as follows:

xseq	::=	x	singleton sequence
	|		empty sequence
	|	($x_1, \cdots, x_n$)	sequence, $n \geq 1$

We refer to long identifiers of the form *strid.longx* as *qualified identifiers*. Qualified identifiers provide a way to access sub-components of structures.

2.2 Grammar for Type Expressions

The grammar for type expressions (*ty*) is given in Figure 2.1. As a syntactic convention, function type expressions associate to the right.

The grammar for type expressions allows for expressing function types, types containing type variables, and types constructed from declared types.

ty	::=	ty_1 -> ty_2	function
	|	*tyvar*	type variable
	|	*tyseq longtycon*	type construction

Figure 2.1: Grammar for type expressions (*ty*).

2.3 Types and Type Functions

Elaboration of type expressions relates type expressions to so-called semantic types. Semantic types stem from a mathematical universe of semantic objects, which is given in Figure 2.2.

As for syntactic types, semantic types are divided into function types, type variables, and constructed types. However, because two different type constructors may stand for the same type, we use a notion of type name to model distinction of types; a constructed type $(\tau_1, \cdots, \tau_k)t$ is equal to another constructed type $(\tau'_1, \cdots, \tau'_k)t'$ iff $\tau_i = \tau'_i$, $i = 1..k$, and $t = t'$. To every type name is associated an arity k—the number of arguments the type name takes. If t is a type name with arity k, we write arity $t = k$.

For any semantic object A, tynames A and tyvars A denote free type names in A and free type variables in A, respectively.

A type function $\theta = \Lambda\alpha^{(k)}.\tau$ has arity k; it must be *closed* (i.e., tyvars$(\tau) \subseteq \alpha^{(k)}$) and the bound variables must be distinct. Two type functions are considered equal if they differ only in their choice of bound variables (alpha-conversion). If t has arity k then we write t to mean $\Lambda\alpha^{(k)}.\alpha^{(k)}t$ (eta-conversion), thus TyName $\subseteq$ TypeFcn. We write the application of a type function θ to a vector $\tau^{(k)}$ of types as $\tau^{(k)}\theta$. If $\theta = \Lambda\alpha^{(k)}.\tau$ we set $\tau^{(k)}\theta = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ (beta-conversion). We write $\tau\{\theta^{(k)}/t^{(k)}\}$ for the result of substituting type functions $\theta^{(k)}$ for type names $t^{(k)}$ in τ and we assume all beta-conversions be carried out after substitution.

2.4 Elaboration of Type Expressions

The rules for elaborating type expressions allow inferences among sentences of the form

$$E \vdash ty \Rightarrow \tau$$

$$
\begin{array}{rcl}
t & \in & \mathrm{TyName}^{(k)} \\
is & \in & \mathrm{IdStatus} = \{\mathtt{v}, \mathtt{c}\} \\
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \cup \mathrm{FunType} \cup \mathrm{ConsType} \\
\tau^{(k)} & \in & \mathrm{Type}^k \\
\alpha^{(k)} & \in & \mathrm{TyVar}^k \\
\tau \to \tau' & \in & \mathrm{Type} \times \mathrm{Type} \\
 & & \mathrm{ConsType} = \cup_{k \geq 0}\ \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t & \in & \mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ or } \Lambda\alpha^{(k)}.\tau & \in & \mathrm{TypeFcn} = \cup_{k \geq 0}\ \mathrm{TyVar}^k \times \mathrm{Type} \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau & \in & \mathrm{TypeScheme} = \cup_{k \geq 0}\ \mathrm{TyVar}^k \times \mathrm{Type} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\text{fin}} \mathrm{Env} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\text{fin}} \mathrm{TypeFcn} \times \mathrm{ValEnv} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \xrightarrow{\text{fin}} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
\Sigma \text{ or } (T)E & \in & \mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Env}
\end{array}
$$

Figure 2.2: Semantic objects for elaborating ModML Core.

where ty is a type expression, E is an environment, and τ is a type. The environment E provides assumptions for free long type constructors of ty. Sentences of this form are read "ty elaborates to τ in E.". The rules for elaborating type expressions are similar to the rules for elaborating type expressions in SML'97.

Type expressions $\boxed{E \vdash ty \Rightarrow \tau}$

$$\frac{E \vdash ty_1 \Rightarrow \tau_1 \quad E \vdash ty_2 \Rightarrow \tau_2}{E \vdash ty_1 \texttt{ -> } ty_2 \Rightarrow \tau_1 \to \tau_2} \tag{2.1}$$

$$\frac{tyvar = \alpha}{E \vdash tyvar \Rightarrow \alpha} \tag{2.2}$$

$$\frac{\begin{array}{c} E(longtycon) = (\theta^{(k)}, VE) \\ tyseq = ty_1 \cdots ty_k \quad E \vdash ty_i \Rightarrow \tau_i,\ i = 1..k \end{array}}{E \vdash tyseq\ longtycon \Rightarrow (\tau_1, \cdots, \tau_k)\theta^{(k)}} \tag{2.3}$$

Comment:

(2.3) The result of elaborating type expressions must be a type, thus, it may be necessary to carry out beta-conversions on the result to fulfil this requirement.

2.5 Grammar for Core

The grammar for the Core language of ModML is shown in Figure 2.3; it gives productions for Core-level expressions (*exp*) and for Core-level declarations (*dec*). As a syntactic convention, function applications associate to the left.

exp	::=	*longvid*	value identifier
	|	`fn` *longvid* `=>` *exp*	function
	|	exp_1 exp_2	application
	|	`let` *dec* `in` *exp* `end`	local declaration
dec	::=	`val` *vid* `=` *exp*	value
	|	`datatype` *tyvarseq tycon* `=` *vid*	datatype
	|	`type` *tyvarseq tycon* `=` *ty*	type
	|	`open` *longstrid*	open

Figure 2.3: Grammar for Core-level expressions (*exp*) and for Core-level declarations (*dec*).

The Core language provides two mechanisms for accessing components in other structures—through qualified identifiers and through the `open` declaration, which makes accessible (non-qualified) all identifiers in the domain of a structure.

For simplicity, the Core language does not support sequential declarations. However, local sequential declarations may be modelled by use of nested `let` expressions. Moreover, the Modules language that we present in Section 2.14 does support sequential declarations. Also for simplicity, a datatype declaration in ModML allows for the declaration of only one nullary value constructor. This simple form of datatypes encapsulates the problems caused by generativity and identifiers with constructor status. It is easy to extend ModML to support datatypes with unary constructors (i.e., constructors that take value arguments) and with multiple constructors.

2.6 Type Schemes and Signatures

A *substitution* S is a finite map from type variables to types. When S is a substitution, we write tynames S to denote the type names that occur free in the range of S. Moreover, the *involved* type variables of a substitution S, written Inv S, is defined by

$$\text{Inv } S = \text{Dom } S \cup (\bigcup_{\alpha \in \text{Dom } S} \text{tyvars}(S(\alpha)))$$

By natural extension, substitutions can be applied to any semantic object that do not bind type names; their effect is to replace each type variable α by $S(\alpha)$. In applying S to a type scheme $\forall\alpha^{(k)}.\tau$, first bound type variables must be changed such that tyvars $\alpha^{(k)} \cap \text{Inv } S = \emptyset$.

A type scheme $\sigma = \forall\alpha^{(k)}.\tau$ *generalises* a type τ', written $\sigma \succ \tau'$, if there exist types $\tau^{(k)}$ such that $\tau' = \tau\{\tau^{(k)}/\alpha^{(k)}\}$. If $\sigma' = \forall\beta^{(l)}.\tau'$ then σ *generalises* σ', written $\sigma \succ \sigma'$, if $\sigma \succ \tau'$ and $\beta^{(l)}$ contains no free type variables of σ. Two type schemes are considered equal if they can be obtained from each other by renaming and reordering of bound variables, and deleting type variables from the prefix which do not occur in the body. We consider a type τ to be a type scheme, identifying it with $\forall().\tau$. It is easy to verify that type scheme generalisation is reflexive and transitive. Moreover, type scheme generalisation is closed under substitution.

Signatures play the rôle at the Modules-level as type schemes do at the Core-level (see Figure 2.2). Similarly, as for type schemes, the prefix (T) in

signatures binds type names. Two signatures are considered equal if they can be obtained from each other by renaming of bound names and by deletion of type names from the prefix that do not occur in the body. When bound type names are changed, we demand that arities of type names are preserved.

As an example, let t and t' be type names with arity 1, let `a` be a type constructor, and let `A` be a value identifier. Then, the signature $(\{t, t'\})(\{\texttt{a} \mapsto (t,\ VE)\},\ VE)$, where $VE = \{\texttt{A} \mapsto (\forall\alpha.\alpha\ t, \texttt{c})\}$, is equal to the signature $(\{t'\})(\{\texttt{a} \mapsto (t',\ VE')\},\ VE')$, where $VE' = \{\texttt{A} \mapsto (\forall\alpha.\alpha\ t', \texttt{c})\}$.

2.7 Elaboration of Core

The rules for elaborating Core-level expressions allow inferences among sentences of the form

$$E \vdash exp \Rightarrow \tau$$

where *exp* is an expression, τ is a type, and E is an environment, which provides assumptions for free identifiers in *exp*. Sentences of this form are read "*exp* elaborates to τ in E."

Expressions $\boxed{E \vdash exp \Rightarrow \tau}$

$$\frac{E(longvid) = (\sigma, is) \quad \sigma \succ \tau}{E \vdash longvid \Rightarrow \tau} \tag{2.4}$$

$$\frac{\begin{array}{c} vid \notin \text{Dom } E \ \text{ or } \ is \text{ of } E(vid) = \texttt{v} \\ E + \{vid \mapsto (\tau, \texttt{v})\} \vdash exp \Rightarrow \tau' \end{array}}{E \vdash \texttt{fn}^{\texttt{v}}\ vid\ \texttt{=>}\ exp \Rightarrow \tau \to \tau'} \tag{2.5}$$

$$\frac{E(longvid) = (\sigma, \texttt{c}) \quad \sigma \succ \tau \quad E \vdash exp \Rightarrow \tau'}{E \vdash \texttt{fn}^{\texttt{c}}\ longvid\ \texttt{=>}\ exp \Rightarrow \tau \to \tau'} \tag{2.6}$$

$$\frac{E \vdash exp_1 \Rightarrow \tau' \to \tau \quad E \vdash exp_2 \Rightarrow \tau'}{E \vdash exp_1\ exp_2 \Rightarrow \tau} \tag{2.7}$$

$$\frac{E \vdash dec \Rightarrow (T)E' \quad E + E' \vdash exp \Rightarrow \tau \quad T \cap (\text{tynames}(E, \tau)) = \emptyset}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{2.8}$$

Comments:

(2.5) and (2.6) In Chapter 4, we shall need to refer to those identifiers that occur free in an expression. In the case of the `fn` expression, this information cannot be obtained from the expression alone; thus, we annotate a `fn` expression by the identifier status for the identifier *vid* (or *longvid*), during elaboration.

(2.8) The side condition here requires that local type names do not escape; without it, the typing rules become unsound [Kah93]. The side condition is different from that of rule (4) of [MTHM97]. In the static semantics of [MTHM97], expressions are elaborated in contexts that, besides from an environment, also holds a set of type names. The side condition then suggests that all type names occurring in the result of elaborating the declaration should occur in the type name set component of the context in which the expression is elaborated. This requirement makes elaboration of a phrase dependent on a larger part of the context than the part that provides assumptions for free identifiers of the phrase. Thus, the static semantics of [MTHM97] is ill-suited for separate compilation. In particular, the separate compilation system that we shall discuss in Chapter 6 assumes that elaboration of a phrase depends only on assumptions for free identifiers of the phrase. The static semantics that we present here has this property (in a sense that we make precise in Chapter 4.)

The rules for elaborating Core-level declarations allow inferences among sentences of the form

$$E \vdash dec \Rightarrow \Sigma$$

where *dec* is a declaration, Σ is a signature, and E is an environment, which provides assumptions for free identifiers of *dec*. Sentences of this form are read "*dec* elaborates to Σ in E." Intuitively, type names that are bound in the resulting signature are those type names that must be considered fresh. In the case of elaboration of expressions, such bound type names stem from datatype declarations; incompatibility of bound type names with other type names is explicitly enforced (by side conditions) in rules that refer to the body of the signature (e.g., rule 2.30).

Declarations $\boxed{E \vdash dec \Rightarrow \Sigma}$

$$\frac{E \vdash exp \Rightarrow \tau \quad \text{tyvars } \alpha^{(k)} \cap \text{tyvars } E = \emptyset}{E \vdash \texttt{val } vid \texttt{ = } exp \Rightarrow (\emptyset)\{vid \mapsto (\forall\alpha^{(k)}.\tau, \texttt{v})\}} \tag{2.9}$$

$$\frac{\begin{array}{c} tyvarseq = \alpha^{(k)} \quad \text{arity } t = k \\ VE = \{vid \mapsto (\forall\alpha^{(k)}.\alpha^{(k)}t, \texttt{c})\} \quad TE = \{tycon \mapsto (t, VE)\} \end{array}}{E \vdash \texttt{datatype } tyvarseq\ tycon \texttt{ = } vid \Rightarrow (\{t\})(TE, VE)} \tag{2.10}$$

$$\frac{tyvarseq = \alpha^{(k)} \quad E \vdash ty \Rightarrow \tau}{E \vdash \texttt{type } tyvarseq\ tycon \texttt{ = } ty \Rightarrow (\emptyset)\{tycon \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}} \tag{2.11}$$

$$\frac{E(longstrid) = E' \quad \text{Dom } E' = I}{E \vdash \texttt{open}^I\ longstrid \Rightarrow (\emptyset)E'} \tag{2.12}$$

Comment:

(2.12) In Chapter 4, we shall need to refer to those identifiers that are declared by a Core-level declaration. In the case of the `open` declaration, this information cannot be obtained from the declaration alone; thus, we annotate the `open` declaration by the set of declared identifiers, during elaboration.

2.8 Grammar for Signature Expressions

The grammar for specifications (*spec*) and signature expressions (*sigexp*) is given in Figure 2.4. For simplicity, ModML does not have support for signature declarations; it is straightforward, however, to extend ModML to support signature declarations. Also for simplicity, ModML does not support type sharing specifications; as we shall see in Section 2.13, type sharing specifications are easily added to the language.

sigexp	::=	`sig` *spec* `end`	basic
	|	*sigexp* `where type`	
		tyvarseq longtycon `=` *ty*	type realisation
spec	::=	`val` *vid* `:` *ty*	value
	|	`type` *tyvarseq tycon*	type
	|	`datatype` *tyvarseq tycon* `=` *vid*	datatype
	|	`structure` *strid* `:` *sigexp*	structure
	|	$spec_1$ $spec_2$	sequential
	|	ε	empty

Figure 2.4: Grammar for specifications (*spec*) and signature expressions (*sigexp*).

$$
\begin{aligned}
\Phi \text{ or } (T)(E, (T')E') &\in \text{FunSig} = \text{TyNameSet} \times (\text{Env} \times \text{Sig}) \\
F &\in \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunSig} \\
B \text{ or } (F, E) &\in \text{StatBasis} = \text{FunEnv} \times \text{Env} \\
(T)B &\in \text{ProgSig} = \text{TyNameSet} \times \text{Basis}
\end{aligned}
$$

Figure 2.5: Semantic objects for elaborating ModML Modules.

2.9 Semantic Objects for Modules

Additional semantic objects for elaborating ModML Modules are given in Figure 2.5.

As for signatures the prefix (T) in functor signatures Φ and program signatures $(T)B$ binds names. Two functor signatures (or program signatures) are considered equal if they can be obtained from each other by renaming of bound names and by deletion of type names from the prefix that do not occur in the body. When bound type names are changed, we demand that arities of type names are preserved.

2.10 Realisation

A *realisation* is a map φ : TyName $\to$ TypeFcn such that t and $\varphi(t)$ have the same arity. The *support* Supp φ of a realisation φ is the set of type names t for which $\varphi(t) \neq t$. The *yield* Yield φ of a realisation φ is the set of type names which occur in some $\varphi(t)$ for which $t \in$ Supp φ. Realisations φ are extended to apply to all semantic objects; their effect is to replace each type name t by $\varphi(t)$.

As an example, let t be a type name with arity 1 and let t' be a type name with arity 0. Then, the map $\varphi = \{t \mapsto \Lambda\alpha.\alpha \to t'\}$ is a realisation with Supp $\varphi = \{t\}$ and Yield $\varphi = \{t'\}$. Moreover, applying the realisation φ to the type $\tau = t'\ t$ results in the type $\varphi(\tau) = t' \to t'$ (after beta-conversion).

2.11 Well-Formedness

A pair (θ, *VE*) is called a *type structure*. Recall that if t has arity k then we sometimes write t to mean the type function $\Lambda\alpha^{(k)}.\alpha^{(k)}t$. A type structure ($\theta$, *VE*) is *well-formed* if either $\theta = t$, for some type name t, or $VE = \{\}$. Any other semantic object is *well-formed* if all type structures in it are well-formed.

Informally, well-formedness guarantees that datatype declarations are always associated with type names. As an example, assume t is a type name with arity 0, α is a type variable, and A is a value identifier, then, the type structure $(t, \{A \mapsto (t, \mathsf{c})\})$ is well-formed, but the type structure $(\Lambda().t \to t, \{A \mapsto (t, \mathsf{c})\})$ is not.

2.12 Elaboration of Signature Expressions

The rules for elaborating specifications allow inferences among sentences of the form

$$B \vdash \mathit{spec} \Rightarrow \Sigma$$

where *spec* is a specification, Σ is a signature, and B is a basis, which provides assumptions for free identifiers of *spec*. Sentences of this form are read "*spec* elaborates to Σ in B."

Specifications $\boxed{B \vdash \mathit{spec} \Rightarrow \Sigma}$

$$\frac{E \text{ of } B \vdash \mathit{ty} \Rightarrow \tau \quad \alpha^{(k)} = \text{tyvars } \tau}{B \vdash \texttt{val } \mathit{vid} \texttt{ : } \mathit{ty} \Rightarrow (\emptyset)\{\mathit{vid} \mapsto (\forall\alpha^{(k)}.\tau, \texttt{v})\}} \tag{2.13}$$

$$\frac{\mathit{tyvarseq} = \alpha^{(k)} \quad \text{arity } t = k}{B \vdash \texttt{type } \mathit{tyvarseq}\ \mathit{tycon} \Rightarrow (\{t\})\{\mathit{tycon} \mapsto (t, \{\})\}} \tag{2.14}$$

$$\frac{\begin{array}{c}\mathit{tyvarseq} = \alpha^{(k)} \quad \text{arity } t = k \\ \mathit{VE} = \{\mathit{vid} \mapsto (\forall\alpha^{(k)}.\alpha^{(k)}t, \texttt{c})\} \quad \mathit{TE} = \{\mathit{tycon} \mapsto (t, \mathit{VE})\}\end{array}}{B \vdash \texttt{datatype } \mathit{tyvarseq}\ \mathit{tycon} \texttt{ = } \mathit{vid} \Rightarrow (\{t\})(\mathit{TE}, \mathit{VE})} \tag{2.15}$$

$$\frac{B \vdash \mathit{sigexp} \Rightarrow (T)E}{B \vdash \texttt{structure } \mathit{strid} \texttt{ : } \mathit{sigexp} \Rightarrow (T)\{\mathit{strid} \mapsto E\}} \tag{2.16}$$

$$\frac{\begin{array}{c}\text{Dom } E_1 \cap \text{Dom } E_2 = \emptyset \\ B \vdash \mathit{spec}_1 \Rightarrow (T_1)E_1 \quad (T_1 \cup T_2) \cap \text{tynames } B = \emptyset \\ B + E_1 \vdash \mathit{spec}_2 \Rightarrow (T_2)E_2 \quad T_2 \cap (T_1 \cup \text{tynames } E_1) = \emptyset\end{array}}{B \vdash \mathit{spec}_1\ \mathit{spec}_2 \Rightarrow (T_1 \cup T_2)(E_1 + E_2)} \tag{2.17}$$

$$\frac{}{B \vdash \varepsilon \Rightarrow (\emptyset)\{\}} \tag{2.18}$$

Comment:

(2.17) The side conditions here allow type names that occur free in $(T_2)E_2$ to be members of T_1, but type names that occur free in $(T_1)E_1$ cannot be members of T_2. The side condition $(T_1 \cup T_2) \cap \text{tynames } B = \emptyset$ is necessary to avoid capture of type names stemming from B in the resulting signature.

The rules for elaborating signature expressions allow inferences among sentences of the form

$$B \vdash \mathit{sigexp} \Rightarrow \Sigma$$

where *sigexp* is a signature expression, Σ is a signature, and B is a basis, which provides assumptions for free identifiers of *sigexp*. Sentences of this form are read "*sigexp* elaborates to Σ in B."

Signature Expressions $\boxed{B \vdash \mathit{sigexp} \Rightarrow \Sigma}$

$$\frac{B \vdash \mathit{spec} \Rightarrow \Sigma}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow \Sigma} \tag{2.19}$$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (T)E \quad T \cap \text{tynames } B = \emptyset \\ \mathit{tyvarseq} = \alpha^{(k)} \quad E \text{ of } B \vdash \mathit{ty} \Rightarrow \tau \\ E(\mathit{longtycon}) = (t, VE) \quad t \in T \quad \varphi = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \end{array}}{B \vdash \mathit{sigexp}\ \texttt{where type}\ \mathit{tyvarseq}\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty} \Rightarrow (T)(\varphi(E))} \tag{2.20}$$

Comments:

(2.20) The resulting signature $(T)(\varphi(E))$ is equal to the signature $(T \setminus \{t\})(\varphi(E))$ because $t \notin$ tynames τ follows from side conditions and elaboration of *ty*.

(2.20) In the rule for `where type` signature expressions in the static semantics of [MTHM97], the result is required to be well-formed (see Section 2.11 for the definition of well-formedness). Because well-formedness is not enforced in rule 2.20, non-well-formed signatures may be introduced by the rules. However, no real structure (i.e., a structure existing outside of a functor body) can match a non-well-formed signature. (It is not the case that all well-formed signatures can be matched by real structures.) Thus, leaving out the well-formedness requirement in rule 2.20 does not contribute to unsoundness of the static semantics. Now, the reason the requirement of well-formedness is inadequate is that well-formedness is not closed under realisation; thus, if well-formedness of the resulting signature is required in rule 2.20 then we do not have the property that elaboration of specifications and signature expressions is closed under realisation. Because a well-formedness requirement in rule 2.20 restricts only what signature expressions elaborate, we can easily enforce this check in an implementation, without compromising soundness. We return to the issue of well-formedness in Section 5.4 on page 93.

2.13 Type Sharing

Standard ML supports type sharing specifications of the form

$$\mathit{spec}\ \texttt{type sharing}\ \mathit{longtycon}_1\ \texttt{=}\ \mathit{longtycon}_2$$

Type sharing specifications of the above form may be added to ModML with the elaboration rule

$$\frac{\begin{array}{c} B \vdash \mathit{spec} \Rightarrow (T)E \quad \varphi = \{t_1 \mapsto t_2\} \quad \text{arity } t_1 = \text{arity } t_2 \\ E(\mathit{longtycon}_i) = (t_i,\, \mathit{VE}_i),\ i = 1..2 \quad \{t_1, t_2\} \subseteq T \end{array}}{B \vdash \mathit{spec}\ \texttt{type sharing}\ \mathit{longtycon}_1\ \texttt{=}\ \mathit{longtycon}_2 \Rightarrow (T)(\varphi(E))} \tag{2.21}$$

For the discussion in this section, we call the language ModML with type sharing specifications added $\text{ModML}_{\text{sharing}}$. Standard ML also supports include specifications of the form

$$\texttt{include}\ \mathit{sigexp}$$

Include specification can be added to ModML with the elaboration rule

$$\frac{B \vdash \mathit{sigexp} \Rightarrow \Sigma}{B \vdash \texttt{include}\ \mathit{sigexp} \Rightarrow \Sigma} \tag{2.22}$$

We call the language ModML with include specifications added $\text{ModML}_{\text{include}}$. We know of no signatures that can be expressed in $\text{ModML}_{\text{sharing}}$ and not in $\text{ModML}_{\text{include}}$. As an example, consider the signature resulting from elaborating (in the empty basis) the $\text{ModML}_{\text{sharing}}$ signature expression

```
sig type t
    type s
    sharing type t = s
end
```

This signature can also be expressed with the $\text{ModML}_{\text{include}}$ signature expression

```
sig type t
    include sig type s end where type s = t
end
```

2.14 Grammar for Modules

The language of Modules constitutes structure-level expressions (*strexp*), structure-level declarations (*strdec*), and top-level declarations (*topdec*), for which the grammar is given in Figure 2.6.

strexp	::=	`struct` *strdec* `end`	basic
	|	*longstrid*	structure identifier
	|	*strexp* `:` *sigexp*	transparent constraint
	|	*strexp* `:>` *sigexp*	opaque constraint
	|	*funid* `(` *strexp* `)`	functor application
strdec	::=	*dec*	declaration
	|	`structure` *strid* `=` *strexp*	structure
	|	$strdec_1$ $strdec_2$	sequential
	|	ε	empty
topdec	::=	*strdec*	structure declaration
	|	`functor` *funid* `(` *strid* `:` *sigexp* `)` `=` *strexp*	functor
	|	$topdec_1$ $topdec_2$	sequential
	|	ε	empty

Figure 2.6: Grammar for structure-level expressions (*strexp*), structure-level declarations (*strdec*), and top-level declarations (*topdec*).

2.15 Signature Instantiation and Functor Signature Instantiation

Instantiation is a mechanism for hiding implementation details of type components of a structure. Formally, an environment E' is an *instance* of a signature $\Sigma = (T)E$, written $\Sigma \geq E'$, if there exists a realisation φ such that $\varphi(E) = E'$ and Supp $\varphi \subseteq T$.

The notion of instantiation extends to functor signatures. A pair (E, Σ) is called a *functor instance*. Given $\Phi = (T_1)(E_1, \Sigma_1)$, a functor instance (E_2, Σ_2)

is an *instance* of Φ, written $\Phi \geq (E_2, \Sigma_2)$, if there exists a realisation φ such that $\varphi(E_1, \Sigma_1) = (E_2, \Sigma_2)$ and Supp $\varphi \subseteq T_1$.

2.16 Enrichment and Signature Matching

Whereas instantiation allows for hiding implementation details of type components of a structure, enrichment allows for hiding components.

A type structure (θ_1, VE_1) *enriches* another type structure (θ_2, VE_2), written $(\theta_1, VE_1) \succ (\theta_2, VE_2)$, if

1. $\theta_1 = \theta_2$

2. Either $VE_1 = VE_2$ or $VE_2 = \{\}$

Further, let σ and σ' be type schemes and let is_1 and is_2 be members of IdStatus. The pair (σ_1, is_1) *enriches* the pair (σ_2, is_2), written $(\sigma_1, is_1) \succ (\sigma_2, is_2)$, if

1. $\sigma_1 \succ \sigma_2$

2. Either $is_1 = is_2$ or $is_2 = \mathtt{v}$

Finally, an environment $E_1 = (SE_1, TE_1, VE_1)$ *enriches* another environment $E_2 = (SE_2, TE_2, VE_2)$, written $E_1 \succ E_2$, if

1. Dom $SE_1 \supseteq$ Dom SE_2 and $SE_1(strid) \succ SE_2(strid)$ for all $strid \in$ Dom SE_2

2. Dom $TE_1 \supseteq$ Dom TE_2 and $TE_1(tycon) \succ TE_2(tycon)$ for all $tycon \in$ Dom TE_2

3. Dom $VE_1 \supseteq$ Dom VE_2 and $VE_1(vid) \succ VE_2(vid)$ for all $vid \in$ Dom VE_2

Signature matching is the combination of signature instantiation and enrichment. An environment E *matches* a signature Σ iff there exists another environment E' such that $\Sigma \geq E' \prec E$.

2.17 Elaboration of Modules

In this section we present inference rules for elaborating structure-level expressions, structure-level declarations and top-level declarations. The rules for elaborating structure-level expressions and structure-level declarations allow inferences among sentences of the forms

$$B \vdash \mathit{strexp} \Rightarrow \Sigma \quad \text{and} \quad B \vdash \mathit{strdec} \Rightarrow \Sigma$$

where *strexp* is a structure-level expression, *strdec* is a structure-level declaration, Σ is a signature, and B is a basis, which provides assumptions for free identifiers in *strexp* or *strdec*, respectively. Sentences of the former form are read "*strexp* elaborates to Σ in B." Sentences of the latter form are read "*strdec* elaborates to Σ in B."

Structure-Level Expressions $\boxed{B \vdash \mathit{strexp} \Rightarrow \Sigma}$

$$\frac{B \vdash \mathit{strdec} \Rightarrow \Sigma}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow \Sigma} \tag{2.23}$$

$$\frac{B(\mathit{longstrid}) = E}{B \vdash \mathit{longstrid} \Rightarrow (\emptyset)E} \tag{2.24}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow (T)E \quad B \vdash \mathit{sigexp} \Rightarrow \Sigma \\ \Sigma \geq E' \prec E \quad T \cap \text{tynames } B = \emptyset \end{array}}{B \vdash \mathit{strexp}\ :\ \mathit{sigexp} \Rightarrow (T)E'} \tag{2.25}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow (T)E \quad B \vdash \mathit{sigexp} \Rightarrow \Sigma \\ \Sigma \geq E' \prec E \quad T \cap \text{tynames } B = \emptyset \end{array}}{B \vdash \mathit{strexp}\ \texttt{:>}\ \mathit{sigexp} \Rightarrow \Sigma} \tag{2.26}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow (T)E \\ B(\mathit{funid}) \geq (E'', (T')E') \quad E \succ E'' \\ (T \cup T') \cap \text{tynames } B = \emptyset \end{array}}{B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)} \Rightarrow (T \cup T')E'} \tag{2.27}$$

Comments:

(2.25) and (2.26) The side condition ($T \cap \text{tynames } B = \emptyset$) is necessary; without it, the structure expression

```
struct datatype a = A
end : sig type a end where type a = int
```

elaborates, in the basis $\{\texttt{int} \mapsto (t, \{\})\}$, to the signature $(\emptyset)\{\texttt{a} \mapsto (t, \{\texttt{A} \mapsto (t, \texttt{c})\})\}$, which is wrong, because a datatype declaration is supposed to generate a fresh type.

(2.27) Generative type names of the functor argument may propagate to the resulting signature.

Structure-Level Declarations $\boxed{B \vdash \mathit{strdec} \Rightarrow \Sigma}$

$$\frac{E \text{ of } B \vdash \mathit{dec} \Rightarrow \Sigma}{B \vdash \mathit{dec} \Rightarrow \Sigma} \tag{2.28}$$

$$\frac{B \vdash \mathit{strexp} \Rightarrow (T)E}{B \vdash \texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp} \Rightarrow (T)\{\mathit{strid} \mapsto E\}} \tag{2.29}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strdec}_1 \Rightarrow (T_1)E_1 \quad (T_1 \cup T_2) \cap \text{tynames } B = \emptyset \\ B + E_1 \vdash \mathit{strdec}_2 \Rightarrow (T_2)E_2 \quad T_2 \cap (T_1 \cup \text{tynames } E_1) = \emptyset \end{array}}{B \vdash \mathit{strdec}_1\ \mathit{strdec}_2 \Rightarrow (T_1 \cup T_2)(E_1 + E_2)} \tag{2.30}$$

$$\frac{}{B \vdash \varepsilon \Rightarrow (\emptyset)\{\}} \tag{2.31}$$

Comment:

(2.30) Type generativity is enforced by appropriate alpha-conversion.

The rules for elaborating top-level declarations allow inferences among sentences of the form

$$B \vdash topdec \Rightarrow (T)B'$$

where *topdec* is a top-level declaration, $(T)B'$ is a program signature, and B is a basis, which provides assumptions for free identifiers in *topdec*. Sentences of this form are read "*topdec* elaborates to $(T)B'$ in B."

Top-level Declarations $\boxed{B \vdash topdec \Rightarrow (T)B'}$

$$\frac{B \vdash strdec \Rightarrow (T)E}{B \vdash strdec \Rightarrow (T)(\{\}, E)} \tag{2.32}$$

$$\frac{\begin{array}{c} B \vdash sigexp \Rightarrow (T)E \quad T \cap \text{tynames } B = \emptyset \\ B + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma \quad F = \{funid \mapsto (T)(E, \Sigma)\} \end{array}}{B \vdash \texttt{functor}\ funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ strexp \Rightarrow (\emptyset)(F, \{\})} \tag{2.33}$$

$$\frac{\begin{array}{c} B \vdash topdec_1 \Rightarrow (T_1)B_1 \quad (T_1 \cup T_2) \cap \text{tynames } B = \emptyset \\ B + B_1 \vdash topdec_2 \Rightarrow (T_2)B_2 \quad T_2 \cap (T_1 \cup \text{tynames } B_1) = \emptyset \end{array}}{B \vdash topdec_1\ topdec_2 \Rightarrow (T_1 \cup T_2)(B_1 + B_2)} \tag{2.34}$$

$$\frac{}{B \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \{\})} \tag{2.35}$$

Comments:

(2.33) The requirement $(T \cap \text{tynames } B = \emptyset)$ ensures that no accidental sharing is assumed between E and B. Because it is the applications of a functor that generate new type names—and not the declaration of it—the set of bound type names in the resulting program signature is empty.

(2.34) As for sequential structure-level declarations, type generativity is enforced by appropriate alpha-conversion.

Chapter 3

Reasoning about ModML

In this chapter, we demonstrate some properties of the static semantics of ModML. In particular, in Section 3.1, we show that many of the relations between semantic objects are closed under realisation. Moreover, we shall demonstrate that elaboration of signature expressions and of structure-level expressions is closed under realisation.

Similar results have been demonstrated by Milner and Tofte in [MT91] for the static semantics of SML'90. In particular, Milner and Tofte demonstrate that elaboration of signature expressions is closed under realisation [MT91, Theorem 10.1]. However, the inspiration for Theorem 10.1 in [MT91] was to show that there is a systematic way to find structure names and type functions that satisfy sharing specifications in the SML'90 static semantics; namely to choose fresh names whenever possible and then identify different names when it is found from sharing specifications that they ought to be identical. In the static semantics of SML'97 and of ModML, sharing specifications and where type signature expressions are dealt with explicitly in the rules by applying realisations, systematically.

There are other reasons, however, why it is important for the static semantics of SML'97 and of ModML that elaboration of signature expressions is closed under realisation. First, as we shall see in Chapter 5, this property makes it possible to translate programs with opaque signature constraints into programs without opaque signature constraints in such a way that elaboration is preserved (in a sense we make precise in Chapter 5.) Second, the realisation property for elaboration of signature expressions is important so as to demonstrate that elaboration of structure-level expressions is closed under realisation; this property is used in Chapter 8 to demonstrate that

functors may be specialised and translated for each application of the functor in such a way that the resulting program is well-typed (in a sense we make precise in Chapter 8.)

Further, in Section 3.2, we shall see that signatures elaborate to so-called type-explicit signatures; this property of elaboration of signatures is important for signature matching to be unique; that is, given Σ and E, there exists at most one environment E^- such that $\Sigma \geq E^- \prec E$. A global requirement that all objects involved in the proof of some sentence be type explicit was removed from the SML'97 static semantics, because it can be shown that no rules would introduce signatures that are not type-explicit; in Section 3.2, we give the proof of this property for ModML.

3.1 Realisation Closedness Properties

In this section we shall see that generalisation, signature instantiation, functor signature instantiation and enrichment are all closed under realisation. Moreover, we shall demonstrate that elaboration of Core-level declarations, of signature expressions, and of structure-level expressions is also closed under realisation.

3.1.1 Generalisation

Generalisation is closed under realisation. As an example, let t_1, t_{real}, and t_{int} be type names with arity 0 and let t_2 be a type name with arity 1. Further, let φ be the realisation $\{t_1 \mapsto \Lambda\beta.\beta \to t_{\text{int}},\ t_2 \mapsto t_{\text{real}}\}$, σ be the type scheme $\forall\alpha.\alpha \to \alpha\ t_1$ and τ be the type $t_2 \to t_2\ t_1$. From the definition of generalisation it follows that $\sigma \succ \tau$. Moreover, we can derive $\varphi(\sigma) = \forall\alpha.\alpha \to (\alpha \to t_{\text{int}})$ and $\varphi(\tau) = t_{\text{real}} \to (t_{\text{real}} \to t_{\text{int}})$. We can now use the definition of generalisation again to derive $\varphi(\sigma) \succ \varphi(\tau)$, as suggested.

We now show that for any realisation φ, if a type scheme σ generalises a type τ then the type scheme $\varphi(\sigma)$ generalises the type $\varphi(\tau)$.

Proposition 3.1.1 (Type generalisation is closed under realisation) *If $\sigma \succ \tau'$ then $\varphi(\sigma) \succ \varphi(\tau')$ for any realisation φ.*

PROOF Let $\sigma = \forall\alpha^{(k)}.\tau$. From the definition of generalisation, we have $\tau' = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ for some $\tau^{(k)}$. It follows that

$$\varphi(\tau') = \varphi(\tau)\{\varphi(\tau^{(k)})/\alpha^{(k)}\} \tag{3.1}$$

From (3.1) and from the definition of generalisation, we have $\forall\alpha^{(k)}.\varphi(\tau) \succ \varphi(\tau')$. Now, because realisations are closed w.r.t. type variables (i.e., tyvars $\varphi = \emptyset$, for any realisation φ), we have $\varphi(\sigma) \succ \varphi(\tau')$, as required. □

Damas and Milner [DM82] have a lemma expressing that generalisation is closed under substitution of types for type variables; given type schemes σ and σ', if $\sigma \succ \sigma'$ then $S(\sigma) \succ S(\sigma')$ for any substitution S, mapping type variables to type schemes. The following proposition states that generalisation of type schemes also is closed under realisation:

Proposition 3.1.2 (Type scheme generalisation is closed under realisation) *If $\sigma \succ \sigma'$ then $\varphi(\sigma) \succ \varphi(\sigma')$ for any realisation φ.*

PROOF Let $\sigma' = \forall\beta^{(l)}.\tau'$. From the definition of generalisation we have

$$\sigma \succ \tau' \tag{3.2}$$

$$\text{tyvars } \beta^{(l)} \cap \text{tyvars } \sigma = \emptyset \tag{3.3}$$

Now, from (3.2) and Proposition 3.1.1 we have

$$\varphi(\sigma) \succ \varphi(\tau') \tag{3.4}$$

Also, from (3.3) and because tyvars $\varphi = \emptyset$ for any realisation φ, we have

$$\text{tyvars } \beta^{(l)} \cap \text{tyvars } \varphi(\sigma) = \emptyset \tag{3.5}$$

It now follows from (3.4), (3.5), and from the definition of generalisation that $\varphi(\sigma) \succ \forall\beta^{(l)}.\varphi(\tau')$, hence, it follows that we have $\varphi(\sigma) \succ \varphi(\sigma')$, as required. □

3.1.2 Signature Instantiation

Signature instantiation is closed under realisation. As an example, consider the signature $\Sigma = (\{t\})\{\mathtt{x} \mapsto (\alpha\ t')\ t\}$. The environment $E = \{\mathtt{x} \mapsto \alpha\ t' \to \alpha\ t'\}$ is an instance of Σ, because there exists a realisation $\varphi = \{t \mapsto \Lambda\beta.\beta \to \beta\}$ such that $\varphi(\{\mathtt{x} \mapsto (\alpha\ t')\ t\}) = E$ and Supp $\varphi \subseteq \{t\}$. Moreover, let $\varphi' = \{t' \mapsto \Lambda\alpha'.t''\}$ be a realisation. It is easy to check that $\varphi'(E)$ is an instance of $\varphi'(\Sigma)$.

Milner and Tofte demonstrate that instantiation of signatures, in the SML'90 static semantics, is closed under realisation [MT91, Lemma 10.2]. This property also holds for the ModML static semantics:

Proposition 3.1.3 (Signature instantiation is closed under realisation) *If* $\Sigma \geq E$ *then* $\varphi(\Sigma) \geq \varphi(E)$ *for any realisation* φ.

PROOF Let $\Sigma = (T_0)E_0$. From the definition of signature instantiation we have there exists a realisation φ_0 such that (Supp $\varphi_0 \subseteq T_0$) and $\varphi_0(E_0) = E$. After renaming of bound names of Σ we can assume

$$(\text{Supp } \varphi \cup \text{Yield } \varphi) \cap T_0 = \emptyset \tag{3.6}$$

Let φ' be the realisation with support Supp $\varphi' \subseteq T_0$ and values

$$\varphi'(t) = \begin{cases} \varphi(\varphi_0(t)) & \text{if } t \in \text{Supp } \varphi_0 \\ t & \text{otherwise} \end{cases}$$

We then have (Supp $\varphi' \subseteq T_0$) and $\varphi'(\varphi(E_0)) = \varphi(\varphi_0(E_0)) = \varphi(E)$, thus, it follows from the definition of signature instantiation and from (3.6) that we have $\varphi(\Sigma) \geq \varphi(E)$, as required. □

3.1.3 Functor Signature Instantiation

Also instantiation of functor signatures is closed under realisation. The following proposition states this property:

Proposition 3.1.4 (Functor signature instantiation is closed under realisation) *If* $\Phi \geq (E, \Sigma)$ *then* $\varphi(\Phi) \geq \varphi(E, \Sigma)$ *for any realisation* φ.

PROOF Let $\Phi = (T_0)(E_0, \Sigma_0)$. From the definition of functor signature instantiation, we have that there exists a realisation φ_0 such that (Supp $\varphi_0 \subseteq T_0$) and $\varphi_0(E_0, \Sigma_0) = (E, \Sigma)$. After appropriate renaming of bound names of Φ, we have

$$(\text{Supp } \varphi \cup \text{Yield } \varphi) \cap T_0 = \emptyset \tag{3.7}$$

Let φ' be the realisation with support Supp $\varphi' \subseteq T_0$ and values

$$\varphi'(t) = \begin{cases} \varphi(\varphi_0(t)) & \text{if } t \in \text{Supp } \varphi_0 \\ t & \text{otherwise} \end{cases}$$

We then have (Supp $\varphi' \subseteq T_0$) and $\varphi'(\varphi(E_0, \Sigma_0)) = \varphi(\varphi_0(E_0, \Sigma_0)) = \varphi(E, \Sigma)$, thus, it follows from the definition of functor signature instantiation and from (3.7) that we have $\varphi(\Phi) \geq \varphi(E, \Sigma)$, as required. □

3.1.4 Enrichment

Enrichment is closed under realisation. We first show that this holds for enrichment of type structures.

Proposition 3.1.5 (Type structure enrichment is closed under realisation) *If* $(\theta_1, VE_1) \succ (\theta_2, VE_2)$ *then* $\varphi(\theta_1, VE_1) \succ \varphi(\theta_2, VE_2)$ *for any realisation* φ.

PROOF From the definition of enrichment, we have $\theta_1 = \theta_2$ and

$$\text{Either } VE_1 = VE_2 \text{ or } VE_2 = \{\}$$

It follows that we have $\varphi(\theta_1) = \varphi(\theta_2)$ and

$$\text{Either } \varphi(VE_1) = \varphi(VE_2) \text{ or } \varphi(VE_2) = \{\}$$

Thus, from the definition of enrichment, we have $\varphi(\theta_1, VE_1) \succ \varphi(\theta_2, VE_2)$, as required. □

The property also holds for enrichment of value environment entries:

Proposition 3.1.6 (Enrichment of value environment entries is closed under realisation) *If* $(\sigma_1, is_1) \succ (\sigma_2, is_2)$ *then* $\varphi(\sigma_1, is_1) \succ \varphi(\sigma_2, is_2)$ *for any realisation* φ.

PROOF The proof follows directly from Proposition 3.1.2 and from the definition of enrichment. □

Using the latter two propositions, we can now demonstrate that enrichment of environments is closed under realisation:

Proposition 3.1.7 (Enrichment is closed under realisation) *If* $E_1 \succ E_2$ *then* $\varphi(E_1) \succ \varphi(E_2)$ *for any realisation* φ.

PROOF The proof is by induction on the structure of E_2. Let $(SE_1, TE_1, VE_1) = E_1$ and $(SE_2, TE_2, VE_2) = E_2$. From assumptions and the definition of enrichment, we have

1. Dom $SE_1 \supseteq$ Dom SE_2 and $SE_1(strid) \succ SE_2(strid)$ for all $strid \in$ Dom SE_2

2. Dom $TE_1 \supseteq$ Dom TE_2 and $TE_1(tycon) \succ TE_2(tycon)$ for all $tycon \in$ Dom TE_2

3. Dom $VE_1 \supseteq$ Dom VE_2 and $VE_1(vid) \succ VE_2(vid)$ for all $vid \in$ Dom VE_2

For each $strid \in$ Dom SE_2 we can apply induction to get $\varphi(SE_1(strid)) \succ \varphi(SE_2(strid))$, hence, we have

$$\text{Dom}(\varphi(SE_1)) \supseteq \text{Dom}(\varphi(SE_2)) \text{ and } (\varphi(SE_1))(strid) \succ (\varphi(SE_2))(strid) \text{ for all } strid \in \text{Dom}(\varphi(SE_2)) \tag{3.8}$$

Further, we can apply Proposition 3.1.5 for each $tycon \in$ Dom TE_2 to get $\varphi(TE_1(tycon)) \succ \varphi(TE_2(tycon))$, hence, we have

$$\text{Dom}(\varphi(TE_1)) \supseteq \text{Dom}(\varphi(TE_2)) \text{ and } (\varphi(TE_1))(tycon) \succ (\varphi(TE_2))(tycon) \text{ for all } tycon \in \text{Dom}(\varphi(TE_2)) \tag{3.9}$$

Finally, we can apply Proposition 3.1.6 for each $vid \in$ Dom VE_2 to get $\varphi(VE_1(vid)) \succ \varphi(VE_2(vid))$, hence, we have

$$\text{Dom}(\varphi(VE_1)) \supseteq \text{Dom}(\varphi(VE_2)) \text{ and } (\varphi(VE_1))(vid) \succ (\varphi(VE_2))(vid) \text{ for all } vid \in \text{Dom}(\varphi(VE_2)) \tag{3.10}$$

Now, from the definition of enrichment and from (3.8), (3.9), and (3.10), we have $\varphi(E_1) \succ \varphi(E_2)$, as required. □

3.1.5 Type Expressions

No new types are generated during elaboration of a type expression; that is, all type names in the result type occur free in the environment in which the type expression is elaborated. Further, elaboration of type expressions is closed under realisation. These observations are expressed in the following proposition.

Proposition 3.1.8 (Elaboration of type expressions is closed under realisation) *If* $E \vdash ty \Rightarrow \tau$, *then* tynames $\tau \subseteq$ tynames E *and* $\varphi(E) \vdash ty \Rightarrow \varphi(\tau)$ *for any realisation* φ.

PROOF The proof is by induction over the structure of ty and proceeds by case analysis.

CASE $ty = ty_1$ -> ty_2 From assumptions and from rule 2.1, we have

$$E \vdash ty_1 \Rightarrow \tau_1 \tag{3.11}$$
$$E \vdash ty_2 \Rightarrow \tau_2 \tag{3.12}$$
$$\tau = \tau_1 \to \tau_2 \tag{3.13}$$

By applying induction to (3.11) and (3.12), we have

$$\text{tynames } \tau_1 \subseteq \text{tynames } E \text{ and } \varphi(E) \vdash ty_1 \Rightarrow \varphi(\tau_1)$$
$$\text{tynames } \tau_2 \subseteq \text{tynames } E \text{ and } \varphi(E) \vdash ty_2 \Rightarrow \varphi(\tau_2)$$

It follows that we have (tynames $\tau \subseteq$ tynames E) and because $\varphi(\tau) = \varphi(\tau_1) \to \varphi(\tau_2)$ follows from (3.13), we can apply rule 2.1 to get $\varphi(E) \vdash ty_1$ -> $ty_2 \Rightarrow \varphi(\tau)$, as required.

CASE $ty = tyvar$ From assumptions, from rule 2.2, and because $\varphi(\alpha) = \alpha$, for any type variable α, we have $\varphi(E) \vdash tyvar \Rightarrow \varphi(\tau)$, as required.

CASE $ty = tyseq\ longtycon$ Write $tyseq$ in the form $ty_1 \cdots ty_k$. From assumptions and from rule 2.3, we have

$$E(longtycon) = (\theta^{(k)}, VE) \tag{3.14}$$
$$E \vdash ty_i \Rightarrow \tau_i,\ i = 1..k$$
$$\tau = (\tau_1, \cdots, \tau_k)\theta^{(k)} \tag{3.15}$$

By applying induction k times, we have

$$\text{tynames } \tau_i \subseteq \text{tynames } E \text{ and } \varphi(E) \vdash ty_i \Rightarrow \varphi(\tau_i),\ i = 1..k \tag{3.16}$$

From (3.14), we have $(\varphi(E))(longtycon) = \varphi(\theta^{(k)}, VE)$. Further, it follows from (3.15) that we have $\varphi(\tau) = (\varphi(\tau_1), \cdots, \varphi(\tau_k))(\varphi(\theta^{(k)}))$. Thus, from (3.16), we have $\varphi(E) \vdash ty \Rightarrow \varphi(\tau)$, as required. Moreover, from (3.14), we have (tynames $\theta^{(k)} \subseteq$ tynames E), hence, from (3.16) and from (3.15), we have (tynames $\tau \subseteq$ tynames E), as required. □

3.1.6 Core-Level Declarations

When *phrase* is either a Core-level expression or a Core-level declaration, when A is either a type or a signature, and when E is an environment, it is possible to infer sentences of the form $E \vdash \mathit{phrase} \Rightarrow A$, such that a free type name in A is not free in E. For example, let `x` and `y` be distinct value identifiers and let t and t' be distinct type names with arity 0. Then the sentence

$$\{\texttt{y} \mapsto (t, \texttt{v})\} \vdash \texttt{fn x => y} \Rightarrow t' \to t$$

is derivable by use of rule 2.4 and rule 2.5. In this example, the type name t' is not free in the environment $\{\texttt{y} \mapsto (t, \texttt{v})\}$. It is crucial that with any type substituted for t', the preceding sentence is still derivable. Indeed, elaboration of Core-level expressions and of Core-level declarations is closed under realisation:

Proposition 3.1.9 (Elaboration of expressions and declarations is closed under realisation) *Let phrase be either an expression or a declaration and let A be either a type or a signature. If $E \vdash \mathit{phrase} \Rightarrow A$ then $\varphi(E) \vdash \mathit{phrase} \Rightarrow \varphi(A)$ for any realisation φ.*

PROOF By induction over the structure of *phrase*. We show the three interesting cases.

CASE $\mathit{exp} = \mathit{longvid}$ From assumptions and from rule 2.4, we have $E(\mathit{longvid}) = (\sigma, is)$ and $\sigma \succ \tau$ and $E \vdash \mathit{exp} \Rightarrow \tau$. It follows from Proposition 3.1.1 that we have $\varphi(\sigma) \succ \varphi(\tau)$, thus, from rule 2.4, we have $\varphi(E) \vdash \mathit{longvid} \Rightarrow \varphi(\tau)$, as required.

CASE $\mathit{exp} = \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}'\ \texttt{end}$ From assumptions and from rule 2.8, we have

$$E \vdash \mathit{dec} \Rightarrow (T)E' \tag{3.17}$$

$$E + E' \vdash \mathit{exp}' \Rightarrow \tau \tag{3.18}$$

$$T \cap \mathrm{tynames}(E, \tau) = \emptyset \tag{3.19}$$

$$E \vdash \mathit{exp} \Rightarrow \tau$$

By appropriate renaming of bound type names, we can assume

$$T \cap (\mathrm{Supp}\ \varphi \cup \mathrm{Yield}\ \varphi) = \emptyset \tag{3.20}$$

From (3.20) and by applying induction to (3.17), we have

$$\varphi(E) \vdash dec \Rightarrow (T)(\varphi(E')) \tag{3.21}$$

We can now apply induction to (3.18) to get

$$\varphi(E + E') \vdash exp' \Rightarrow \varphi(\tau) \tag{3.22}$$

Moreover, from (3.19) and from (3.20), we have

$$T \cap \text{tynames}(\varphi(E), \varphi(\tau)) = \emptyset \tag{3.23}$$

Now, from rule 2.8 and from (3.21), (3.22), and (3.23), we have $\varphi(E) \vdash exp \Rightarrow \varphi(\tau)$, as required.

CASE $dec =$ `datatype` *tyvarseq tycon* `=` *vid* From assumptions we have $E \vdash dec \Rightarrow \Sigma$. It follows from rule 2.10 and by appropriate renaming of bound names of Σ that we have $\varphi(E) \vdash dec \Rightarrow \varphi(\Sigma)$, as required. □

3.1.7 Specifications and Signature Expressions

We shall now see that elaboration of specifications and of signature expressions is closed under realisation.

Proposition 3.1.10 *Let phrase be either a specification or a signature expression. If* $B \vdash phrase \Rightarrow \Sigma$ *then* tynames $\Sigma \subseteq$ tynames B *and* $\varphi(B) \vdash phrase \Rightarrow \varphi(\Sigma)$ *for any realisation* φ.

PROOF The proof is by induction on the structure of *phrase* and proceeds by case analysis.

CASE $spec =$ `val` *vid* `:` *ty* From assumptions and from rule 2.13, we have

$$E \text{ of } B \vdash ty \Rightarrow \tau \tag{3.24}$$

$$\alpha^{(k)} = \text{tyvars } \tau \tag{3.25}$$

$$\Sigma = (\emptyset)\{vid \mapsto \forall\alpha^{(k)}.\tau\} \tag{3.26}$$

From Proposition 3.1.8 and from (3.24), we have

$$\text{tynames } \tau \subseteq \text{tynames}(E \text{ of } B) \tag{3.27}$$

$$\varphi(E \text{ of } B) \vdash ty \Rightarrow \varphi(\tau) \tag{3.28}$$

Now, from (3.26) and from (3.27), we have

$$\text{tynames } \Sigma \subseteq \text{tynames } B$$

Further, let

$$\beta^{(l)} = \text{tyvars}(\varphi(\tau)) \tag{3.29}$$

From (3.28) and from (3.29), we can now apply rule 2.13 to get

$$\varphi(B) \vdash \texttt{val}\ vid\ :\ ty\ \Rightarrow (\emptyset)\{vid \mapsto \forall\beta^{(l)}.\varphi(\tau)\} \tag{3.30}$$

Now, from (3.25) and because tyvars $\tau \supseteq \text{tyvars}(\varphi(\tau))$ for any type τ and realisation φ, we have $\alpha^{(k)} \supseteq \beta^{(l)}$, hence, from the definition of equality on type schemes, we have $\forall\beta^{(l)}.\varphi(\tau) = \forall\alpha^{(k)}.\varphi(\tau)$. It follows that we have $\varphi(B) \vdash spec \Rightarrow \varphi(\Sigma)$, as required.

CASE $spec = \texttt{type}\ tyvarseq\ tycon$ From assumptions and from rule 2.14, we have

$$\alpha^{(k)} = tyvarseq \tag{3.31}$$

$$\text{arity } t = k \tag{3.32}$$

$$\Sigma = (\{t\})\{tycon \mapsto (t, \{\})\} \tag{3.33}$$

From (3.33), we have tynames $\Sigma = \emptyset$, hence, we have

$$\text{tynames } \Sigma \subseteq \text{tynames } B \tag{3.34}$$

$$\varphi(\Sigma) = \Sigma \tag{3.35}$$

Now, from rule 2.14 and from (3.31), (3.32), (3.33), and (3.35), we have $\varphi(B) \vdash spec \Rightarrow \varphi(\Sigma)$, as required.

CASE $spec = \texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid$ From assumptions and from rule 2.15, we have

$$\alpha^{(k)} = tyvarseq \tag{3.36}$$

$$\text{arity } t = k \tag{3.37}$$

$$\Sigma = (\{t\})(\{tycon \mapsto (t, VE)\}, VE) \tag{3.38}$$

where $VE = \{vid \mapsto \forall\alpha^{(k)}.\alpha^{(k)}t\}$. From (3.38), we have tynames $\Sigma = \emptyset$, hence, we have

$$\text{tynames } \Sigma \subseteq \text{tynames } B$$

$$\varphi(\Sigma) = \Sigma \tag{3.39}$$

Now, from rule 2.15 and from (3.36), (3.37), (3.38), and (3.39), we have $\varphi(B) \vdash \mathit{spec} \Rightarrow \varphi(\Sigma)$, as required.

CASE $\mathit{spec} = \texttt{structure}\ \mathit{strid} : \mathit{sigexp}$ From assumptions and from rule 2.16, we have

$$B \vdash \mathit{sigexp} \Rightarrow (T)E \tag{3.40}$$
$$\Sigma = (T)\{\mathit{strid} \mapsto E\} \tag{3.41}$$

We can immediately apply induction to (3.40) to get

$$\text{tynames}((T)E) \subseteq \text{tynames}\ B \tag{3.42}$$
$$\varphi(B) \vdash \mathit{sigexp} \Rightarrow \varphi((T)E) \tag{3.43}$$

It follows from (3.42) that we have

$$\text{tynames}\ \Sigma \subseteq \text{tynames}\ B$$

By renaming of bound names of Σ, we can assume

$$T \cap (\text{Supp}\ \varphi \cup \text{Yield}\ \varphi) = \emptyset$$

hence, from (3.41), from rule 2.16, and from (3.43), we have $\varphi(B) \vdash \mathit{spec} \Rightarrow \varphi(\Sigma)$, as required.

CASE $\mathit{spec} = \mathit{spec}_1\ \mathit{spec}_2$ From assumptions and from rule 2.17, we have

$$B \vdash \mathit{spec}_1 \Rightarrow (T_1)E_1 \tag{3.44}$$
$$(T_1 \cup T_2) \cap \text{tynames}\ B = \emptyset \tag{3.45}$$
$$B + E \vdash \mathit{spec}_2 \Rightarrow (T_2)E_2 \tag{3.46}$$
$$T_2 \cap (T_1 \cup \text{tynames}\ E_1) = \emptyset \tag{3.47}$$
$$\Sigma = (T_1 \cup T_2)(E_1 + E_2) \tag{3.48}$$

By renaming of bound names of Σ, we can assume

$$(T_1 \cup T_2) \cap (\text{Supp}\ \varphi \cup \text{Yield}\ \varphi) = \emptyset \tag{3.49}$$

By applying induction twice to (3.44) and (3.46) we get

$$\text{tynames}((T_1)E_1) \subseteq \text{tynames}\ B \tag{3.50}$$
$$\varphi(B) \vdash \mathit{spec}_1 \Rightarrow \varphi((T_1)E_1) \tag{3.51}$$
$$\text{tynames}((T_2)E_2) \subseteq \text{tynames}(B + E_1) \tag{3.52}$$
$$\varphi(B + E_1) \vdash \mathit{spec}_2 \Rightarrow \varphi((T_2)E_2) \tag{3.53}$$

From (3.48), (3.50), (3.52), and (3.45), we have

$$\text{tynames } \Sigma \subseteq \text{tynames } B$$

Now, from (3.49), (3.45), and (3.47), we have

$$(T_1 \cup T_2) \cap \text{tynames}(\varphi(B)) = \emptyset \tag{3.54}$$
$$T_2 \cap (T_1 \cup \text{tynames}(\varphi(E_1)) = \emptyset \tag{3.55}$$

From rule 2.17 and from (3.51), (3.54), (3.53), (3.55), (3.49), and (3.48), we have $\varphi(B) \vdash \mathit{spec} \Rightarrow \varphi(\Sigma)$, as required.

CASE $\mathit{spec} = \varepsilon$ The result follows immediately from rule 2.18.

CASE $\mathit{sigexp} =$ `sig` spec `end` The result follows immediately from rule 2.19 by induction.

CASE $\mathit{sigexp} = \mathit{sigexp}'$ `where type` $\mathit{tyvarseq}\ \mathit{longtycon}$ `=` ty From assumptions and from rule 2.20, we have

$$B \vdash \mathit{sigexp}' \Rightarrow (T)E \tag{3.56}$$
$$T \cap \text{tynames } B = \emptyset \tag{3.57}$$
$$\mathit{tyvarseq} = \alpha^{(k)} \tag{3.58}$$
$$E \text{ of } B \vdash \mathit{ty} \Rightarrow \tau \tag{3.59}$$
$$E(\mathit{longtycon}) = (t, VE) \tag{3.60}$$
$$t \in T \tag{3.61}$$
$$\varphi' = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \tag{3.62}$$
$$\Sigma = (T)(\varphi'(E)) \tag{3.63}$$

By renaming of bound names of Σ, we can assume

$$T \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{3.64}$$

We can now apply induction to (3.56) to get

$$\text{tynames}((T)E) \subseteq \text{tynames } B \tag{3.65}$$
$$\varphi(B) \vdash \mathit{sigexp}' \Rightarrow \varphi((T)E)) \tag{3.66}$$

Further, from (3.59) and from Proposition (3.1.8), we have

$$\text{tynames } \tau \subseteq \text{tynames } B \tag{3.67}$$
$$\varphi(E \text{ of } B) \vdash \mathit{ty} \Rightarrow \varphi(\tau) \tag{3.68}$$

Now, from (3.65), (3.62), (3.63), and (3.67), we have

$$\text{tynames } \Sigma \subseteq \text{tynames } B$$

Moreover, from (3.60), (3.61), and (3.64), we have

$$(\varphi(E))(longtycon) = (t, \varphi(VE)) \tag{3.69}$$

From (3.57) and from (3.64), we have

$$T \cap \text{tynames}(\varphi(B)) = \emptyset$$

Now, let

$$\varphi'' = \{t \mapsto \Lambda\alpha^{(k)}.\varphi(\tau)\} \tag{3.70}$$

From rule 2.20 and from (3.66), (3.58), (3.68), (3.69), (3.61), and (3.70), we have

$$\varphi(B) \vdash sigexp \Rightarrow \Sigma'$$

where $\Sigma' = (T)(\varphi''(\varphi(E)))$. From (3.61), (3.64), (3.62), and (3.70), we have $\varphi''(\varphi(E)) = \varphi(\varphi'(E))$, thus, from (3.63) and from (3.64), we have $\varphi(B) \vdash sigexp \Rightarrow \varphi(\Sigma)$, as required. □

3.1.8 Structure-Level Expressions

In this section, we shall see that elaboration of structure-level expressions and of structure-level declarations is closed under realisation.

Proposition 3.1.11 *Let phrase be either a structure-level expression or a structure-level declaration. If $B \vdash phrase \Rightarrow \Sigma$ then $\varphi(B) \vdash phrase \Rightarrow \varphi(\Sigma)$ for any realisation φ.*

PROOF The proof is by induction over the structure of *strexp* and *strdec*.

CASE $strexp = strexp' : sigexp$ Let φ be any realisation. From assumptions and from rule 2.25, we have

$$B \vdash strexp' \Rightarrow (T)E \tag{3.71}$$

$$B \vdash sigexp \Rightarrow \Sigma \tag{3.72}$$

$$\Sigma \geq E' \prec E \tag{3.73}$$

$$T \cap \text{tynames } B = \emptyset \tag{3.74}$$

$$B \vdash strexp \Rightarrow (T)E' \tag{3.75}$$

By appropriate renaming of bound names, we have

$$T \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{3.76}$$

By applying induction to (3.71) and from (3.76), we have

$$\varphi(B) \vdash \mathit{strexp}' \Rightarrow (T)(\varphi(E)) \tag{3.77}$$

From Proposition 3.1.10 and from (3.72), we have

$$\varphi(B) \vdash \mathit{sigexp} \Rightarrow \varphi(\Sigma) \tag{3.78}$$

From Proposition 3.1.3, from Proposition 3.1.7, and from (3.73), we have

$$\varphi(\Sigma) \geq \varphi(E') \prec \varphi(E) \tag{3.79}$$

Moreover, from (3.76) and (3.74), we have

$$T \cap \text{tynames } \varphi(B) = \emptyset \tag{3.80}$$

Now, from rule 2.25 and from (3.77), (3.78), (3.79), (3.80), and (3.76), we have $\varphi(B) \vdash \mathit{strexp} \Rightarrow \varphi((T)E')$ for any realisation φ, as required.

CASE $\mathit{strexp} = \mathit{strexp}'$:> sigexp The proof for this case is similar to the proof for transparent signature constraints.

CASE $\mathit{strexp} = \mathit{funid}$ (strexp) Let φ be any realisation. From assumptions and from rule 2.27, we have

$$B \vdash \mathit{strexp}' \Rightarrow (T)E \tag{3.81}$$
$$B(\mathit{funid}) \geq (E'', (T')E') \tag{3.82}$$
$$E \succ E'' \tag{3.83}$$
$$(T \cup T') \cap \text{tynames } B = \emptyset \tag{3.84}$$
$$B \vdash \mathit{strexp} \Rightarrow (T \cup T')E' \tag{3.85}$$

By appropriate renaming of bound names, we can assume

$$(T \cup T') \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{3.86}$$

By applying induction to (3.81) and from (3.86), we have

$$\varphi(B) \vdash \mathit{strexp}' \Rightarrow (T)(\varphi(E)) \tag{3.87}$$

From Proposition 3.1.4, from Proposition 3.1.7, and from (3.82), (3.83), and (3.86), we have

$$(\varphi(B))(\mathit{funid}) \geq (\varphi(E''), (T')(\varphi(E'))) \tag{3.88}$$
$$\varphi(E) \succ \varphi(E'') \tag{3.89}$$

From (3.84) and from (3.86), we have

$$(T \cup T') \cap \text{tynames } \varphi(B) = \emptyset \tag{3.90}$$

Now, from rule 2.27 and from (3.87), (3.88), (3.89), (3.90), and (3.86), we have $\varphi(B) \vdash \mathit{strexp} \Rightarrow \varphi((T \cup T')E')$ for any realisation φ, as required.

CASE $\mathit{strdec} = \mathit{dec}$ The required result follows directly from Proposition 3.1.9.

CASE $\mathit{strdec} = \mathit{strdec}_1\ \mathit{strdec}_2$ Let φ be any realisation. From assumptions and from rule 2.30, we have

$$B \vdash \mathit{strdec}_1 \Rightarrow (T_1)E_1 \tag{3.91}$$
$$(T_1 \cup T_2) \cap \text{tynames } B = \emptyset \tag{3.92}$$
$$B + E_1 \vdash \mathit{strdec}_2 \Rightarrow (T_2)E_2 \tag{3.93}$$
$$T_2 \cap (T_1 \cup \text{tynames } E_1) = \emptyset \tag{3.94}$$
$$B \vdash \mathit{strdec} \Rightarrow (T_1 \cup T_2)(E_1 + E_2) \tag{3.95}$$

By appropriate renaming of bound names, we can assume

$$(T_1 \cup T_2) \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{3.96}$$

By applying induction twice to (3.91) and (3.93) and from (3.96), we have

$$\varphi(B) \vdash \mathit{strdec}_1 \Rightarrow (T_1)(\varphi(E_1)) \tag{3.97}$$
$$\varphi(B) + \varphi(E_1) \vdash \mathit{strdec}_2 \Rightarrow (T_2)(\varphi(E_2)) \tag{3.98}$$

From (3.96) and from (3.92) and (3.94), we have

$$(T_1 \cup T_2) \cap \text{tynames } \varphi(B) = \emptyset \tag{3.99}$$
$$T_2 \cap (T_1 \cup \text{tynames } \varphi(E_1)) = \emptyset \tag{3.100}$$
$$\tag{3.101}$$

Now, from rule 2.30 and from (3.97), (3.99), (3.98), (3.100), and (3.96), we have $\varphi(B) \vdash \mathit{strdec} \Rightarrow \varphi((T_1 \cup T_2)(E_1 + E_2))$ for any realisation φ, as required.

Each of the proofs for the remaining cases either follows directly or follows directly by induction. □

3.2 Type-Explication

Recall from Section 2.16 that an environment E *matches* a signature $\Sigma_1 = (T_1)E_1$ if there exists an environment E^- such that $\Sigma_1 \geq E^- \prec E$. Thus, matching is the combination of enrichment and instantiation. It is crucial that, given Σ_1 and E, there is at most one such E^-. If there were more than one such E^- then elaboration of transparent signature constraints (rule 2.25) and functor applications (rule 2.27) would become non-deterministic. The following notion of type-explication is sufficient to ensure that signature matching is unique:

Definition 3.2.1 (Type-explication) A signature $(T)E$ is *type-explicit* if for every $t \in T$ there exists a *longtycon* such that θ of $(E(\mathit{longtycon})) = t$ (assuming $T \subseteq$ tynames E.) Moreover, a functor environment is *type-explicit* if for every functor signature $(T)(E, \Sigma)$ in its range, $(T)E$ is type-explicit. □

In the static semantics of SML'90 it is required that, whenever a signature expression elaborates to a signature, the signature be type-explicit [MTH90, rule 65]. Type-explication is not explicitly enforced in the rules for the static semantics of SML'97 or for ModML; it can be shown, both for SML'97 and for ModML, that signature expressions always elaborate to type-explicit signatures. We give the proof here for ModML:

Proposition 3.2.2 *Let phrase be either a specification or a signature expression. If* $B \vdash \mathit{phrase} \Rightarrow \Sigma$ *then* Σ *is type-explicit.*

PROOF The proof is by induction over the structure of *phrase*. There are only two interesting cases.

CASE $\mathit{spec} = \mathit{spec}_1\ \mathit{spec}_2$ From assumptions and from rule 2.17, we have

$$B \vdash \mathit{spec}_1 \Rightarrow (T_1)E_1 \tag{3.102}$$

$$B + E_1 \vdash spec_2 \Rightarrow (T_2)E_2 \tag{3.103}$$
$$\Sigma = (T_1 \cup T_2)(E_1 + E_2) \tag{3.104}$$
$$\text{Dom } E_1 \cap \text{Dom } E_2 = \emptyset \tag{3.105}$$

By applying induction twice to (3.102) and (3.103) we get that both $(T_1)E_1$ and $(T_2)E_2$ are type-explicit. From the definition of type-explication and from (3.105), we have Σ is type-explicit, as required.

CASE $sigexp = sigexp'$ `where type` $tyvarseq\ longtycon$ `=` ty From assumptions and from rule 2.20, we have

$$B \vdash sigexp' \Rightarrow (T)E \tag{3.106}$$
$$T \cap \text{tynames } B = \emptyset \tag{3.107}$$
$$tyvarseq = \alpha^{(k)}$$
$$E \text{ of } B \vdash ty \Rightarrow \tau \tag{3.108}$$
$$t \in T \tag{3.109}$$
$$\varphi = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \tag{3.110}$$
$$\Sigma = (T)(\varphi(E)) \tag{3.111}$$

By applying induction to (3.106), we have

$$(T)E \text{ is type-explicit} \tag{3.112}$$

Further, from Proposition 3.1.8 and from (3.108), we have

$$\text{tynames } \tau \subseteq \text{tynames } B \tag{3.113}$$

From (3.107), (3.113), (3.109), and (3.110), we have $t \notin$ tynames $(\varphi(E))$, hence, from (3.111) and from the definition of equality of signatures, we have

$$\Sigma = (T \setminus \{t\})(\varphi(E)) \tag{3.114}$$

Now, from (3.112), from (3.114), from (3.110), and from the definition of type-explication, we have Σ is type-explicit as required. □

Interestingly, Proposition 3.2.2 holds without any use of well-formedness of type structures. The proposition relies in an essential way on the side-conditions in the rules for `where type` signature expressions and on the restriction that sequential specifications cannot have overlapping domains.

The following corollary follows from Proposition 3.2.2 and from the rules 2.32 through 2.35.

Corollary 3.2.3 *If* $B \vdash topdec \Rightarrow (T)(F, E)$ *then* F *is type-explicit.*

Proposition 3.2.2 and Corollary 3.2.3 justify why an explicit side-condition about type-explicitness is not needed in rule 2.19 and in rule 2.20.

Chapter 4

Elaboration Dependence

In this chapter we demonstrate that, for the language ModML, elaboration of a program phrase depends only on the part of the basis that describes free identifiers of the phrase. More precisely, we show that elaboration of a program in a given basis depends on assumptions in the basis for (1) free long structure identifiers, (2) free long type constructors, (3) free long value identifiers, and (4) free functor identifiers. This property is important for separate compilation. Consider the program consisting of the two program units `A` and `B`, where `B` depends on `A`:

```
(* Program unit A *)   structure S = struct
                          val a = 5
                          val b = true
                        end

(* Program unit B *)   val c = (S.b, 2)
```

In this example, program unit `B` refers only to the long value identifier `S.b`. Now, if program unit `A` is modified, but still contains a component `b` of type `bool`, then `B` need not be reelaborated, and depending on properties of compilation, it may not need to be recompiled, either.

In SML'90 [MTH90] elaboration of a program phrase depends not only on assumptions for those identifiers that occur free in the phrase, but also on assumptions for those identifiers that do not occur free in the phrase. In the static semantics for SML'90, when a program phrase *phrase* elaborates in some basis B to a semantic object A then a requirement called admissibility

is enforced on the assembly of objects that occur in the proof of the sentence $B \vdash \mathit{phrase} \Rightarrow A$. Now, assume B' is a larger basis than B (in the sense that B' associates assumptions to more identifiers but agrees with B on assumptions for those identifiers that occur free in *phrase*) then admissibility of the assembly of objects that occur in the proof of the sentence $B' \vdash \mathit{phrase} \Rightarrow A$ cannot be guaranteed. Further, the static semantics of SML'90 also requires that when a signature expression *sigexp* elaborates in some basis B to a signature Σ then Σ must be what is called a principal signature for *sigexp* in B. Now, assume that B' is a smaller basis than B (in the sense that B' associates assumptions to fewer identifiers but agrees with B on assumptions for those identifiers that occur free in *sigexp*) then it cannot be guaranteed that Σ is a principal signature for *sigexp* in B'.[1]

In the static semantics of SML'97 [MTHM97] the notions of admissibility and principal signatures are eliminated. However, because of certain side conditions in the semantic rules and because of the way generativity of type names is enforced, one cannot restrict assumptions as expected. Recall, that in SML'97 a context is a triple (T, U, E) of a set of type names, a set of explicit type variables, and an environment. Elaboration is presented as a set of inferences among sentences of which some are of the form $C \vdash \mathit{phrase} \Rightarrow A$, where C is a context such that tynames(E of C) $\subseteq$ (T of C), *phrase* is a phrase of the Core language and A is some semantic object. Now, in some rules of the static semantics (more precisely, rule 4, rule 14, and rule 26), side conditions require type names in some involved objects to be contained in (T of C). Such inclusion requirements do not work well with restriction of assumptions to those identifiers that occur free in the program phrase. In particular, if a program phrase *phrase* elaborates in some context C to an object A then it is not possible to restrict the type name set component of C and demonstrate that *phrase* elaborates in the restricted context to the object A. In the static semantics of ModML, generativity is modelled by type abstraction and the static semantics is defined in such a way that no inclusion requirements are necessary.

There are two reasons why it is not possible to determine what identifiers occur free in a ModML program phrase without some information about the basis in which the program phrase is elaborated. The first reason is the `open` declaration; it is not possible to determine what identifiers occur free in a

[1] The reason that Σ cannot be guaranteed to be a principal signature for *sigexp* in B' has to do with cover [MTH90, Section 5.13].

program phrase without either restricting the use of the `open` declaration or annotating the `open` construct with information about what identifiers are declared by the declaration. The second reason is the support for simple patterns; it is not always possible, from a program phrase alone, to determine whether an identifier in a pattern is a value constructor or a value variable. In the latter case the pattern binds the identifier, whereas in the former case, the identifier occurs free in the program phrase. By annotating `open` declarations with the set of declared identifiers and `fn` expressions with identifier status information, it becomes possible to determine what identifiers occur free in a program phrase.

The remainder of this chapter is organised as follows. In Section 4.1, we define some operations for calculating with sets of identifiers. In Section 4.2 and Section 4.3, we define the notions of restriction and strong enrichment, respectively. The definitions of restriction and strong enrichment are motivated by the attempt to demonstrate the following property: If a ModML program phrase elaborates under some assumptions B to an object A and if B' strongly enriches the assumptions B, restricted to those long identifiers that occur free in the program phrase, then the program phrase elaborates under assumptions B' to the object A. This property does not hold for ModML program phrases with free long structure identifiers in them. To see why, consider the ModML program

```
open S
val f = a
```

and assume that the program is elaborated to an object A in a basis for which the environment assumed for `S` has no value component `a`. Then, it does not necessarily hold that the program elaborates to the object A in a basis for which the environment assumed for `S` does have a value component `a`. Thus, we define a notion of agreement in Section 4.4; to demonstrate the preceding property, we must further require that B and B' agree on assumptions for those long structure identifiers that occur free in the program phrase. Moreover, we must also require that B and B' agree on those value identifiers (in the domains of B and B') with constructor identifier status. This last requirement is necessary to deal correctly with identifiers in patterns.

In Section 4.5 through Section 4.7, we demonstrate elaboration dependence for ModML program phrases. Finally, in Section 4.8, we suggest some possibilities for refining elaboration dependence.

4.1 Identifiers

We have earlier grouped identifiers into classes. We now extend this classification and classify an identifier as *non-qualified*, if it is in the set

$$\mathrm{Id} = \mathrm{FunId} \cup \mathrm{StrId} \cup \mathrm{TyCon} \cup \mathrm{VId}$$

Further, we classify an identifier as *long*, if it is in the set

$$\mathrm{LongId} = \mathrm{FunId} \cup \mathrm{LongTyCon} \cup \mathrm{LongStrId} \cup \mathrm{LongVId}$$

We use *longx* to range over long identifiers. The set of non-qualified identifiers is a subset of the set of long identifiers.

The *curtailment* of a set of long identifiers L to a set of non-qualified identifiers I, written $L \setminus\setminus I$, is the set $L \setminus (I \cup \{strid.longx \mid strid \in I\})$.

Further, when L is a set of long identifiers, the *qualifier* of L, written $\mathrm{Qual}(L)$, is the set

$$\{strid \mid \exists longtycon \text{ s.t. } strid.longtycon \in L \vee \exists longvid \text{ s.t. } strid.longvid \in L\}$$

Proposition 4.1.1 *If* $L' \subseteq L$ *then* $\mathrm{Qual}(L') \subseteq \mathrm{Qual}(L)$.

PROOF Follows from the definition of qualifier. □

The *projection* of a set of long identifiers L to a structure identifier *strid*, written $L/strid$, is the set

$$\{longtycon \mid strid.longtycon \in L\} \cup \{longvid \mid strid.longvid \in L\}$$

Proposition 4.1.2 *If* $L' \subseteq L$ *and* $strid \in \mathrm{Qual}(L')$ *then* $(L'/strid) \subseteq (L/strid)$.

PROOF From assumptions, we have $L' \subseteq L$ and $strid \in \mathrm{Qual}(L')$. Assume $strid.longx \in L'$ for some *longx*. It suffices to show that if $strid.longx \in L$ then $longx \in (L/strid)$; this follows from the definition of projection. □

We now define a function for extracting, from an environment (or a basis), the set of long identifiers for which assumptions occur in the environment (or basis). The function is defined inductively as follows:

$$\begin{aligned}\mathrm{LongIds}(A) = \; & (\mathrm{Dom}\ A \cap (\mathrm{FunId} \cup \mathrm{TyCon} \cup \mathrm{VId})) \cup \\ & \{strid.longid \mid strid \in (\mathrm{Dom}\ A \cap \mathrm{StrId}) \wedge \\ & \qquad longid \in \mathrm{LongIds}(A(strid))\}\end{aligned}$$

Intuitively, $longx \in \mathrm{LongIds}(A)$ iff *longx* is not a long structure identifier and there exists an object A' such that $A(longx) = A'$.

4.2 Restriction

The *restriction* of a finite map f to a set of identifiers $A \subseteq \mathrm{Dom}\ f$, written $f \downarrow A$, is the finite map with domain A and values $(f \downarrow A)(x) = f(x)$. Restriction is extended to environments as follows. Assume L be a set of long identifiers. The *restriction* of an environment $E = (SE, TE, VE)$ *to* L, written $E \downarrow L$, is the environment (TE', SE', VE'), where

1. $SE' = \{strid \mapsto SE(strid) \downarrow (L/strid) \mid strid \in \mathrm{Qual}(L)\}$
2. $TE' = TE \downarrow (L \cap \mathrm{TyCon})$
3. $VE' = VE \downarrow (L \cap \mathrm{VId})$

Further, the *restriction* of a basis $B = (F, E)$ *to* L, written $B \downarrow L$, is the basis $(F \downarrow (L \cap \mathrm{FunId}), E \downarrow L)$.

Proposition 4.2.1 *If* $(L \setminus (\mathrm{LongStrId} \cup \mathrm{FunId})) \subseteq \mathrm{LongIds}(E)$ *then there exists an environment* E' *such that* $E' = E \downarrow L$.

PROOF By induction over the structure of E. □

4.3 Strong Enrichment

An environment $E_1 = (SE_1, TE_1, VE_1)$ *strongly enriches* another environment $E_2 = (SE_2, TE_2, VE_2)$, written $E_1 \sqsupseteq E_2$, if

1. $\mathrm{Dom}\ SE_1 \supseteq \mathrm{Dom}\ SE_2$ and $SE_1(strid) \sqsupseteq SE_2(strid)$ for all $strid \in \mathrm{Dom}\ SE_2$
2. $\mathrm{Dom}\ TE_1 \supseteq \mathrm{Dom}\ TE_2$ and $TE_1(tycon) = TE_2(tycon)$ for all $tycon \in \mathrm{Dom}\ TE_2$
3. $\mathrm{Dom}\ VE_1 \supseteq \mathrm{Dom}\ VE_2$ and $VE_1(vid) = VE_2(vid)$ for all $vid \in \mathrm{Dom}\ VE_2$

Moreover, a basis $B_1 = (F_1, E_1)$ *strongly enriches* another basis $B_2 = (F_2, E_2)$, written $B_1 \sqsupseteq B_2$, if

1. $\mathrm{Dom}\ F_1 \supseteq \mathrm{Dom}\ F_2$ and $F_1(funid) = F_2(funid)$ for all $funid \in \mathrm{Dom}\ F_2$

2. $E_1 \sqsupseteq E_2$

In the remainder of this section, we demonstrate a series of propositions about the interaction between restriction and strong enrichment; the propositions are used, in the sections to follow, to demonstrate that elaboration of a ModML program phrase depends only on assumptions for those identifiers that occur free in the program phrase. We first show that if E' is the result of restricting an environment E to a set of long identifiers then E strongly enriches E':

Proposition 4.3.1 *If* $E' = E \downarrow L$ *then* $E \sqsupseteq E'$.

PROOF The proof is by induction on the structure of long identifiers in L. Write E' in the form (SE', TE', VE') and write E in the form (SE, TE, VE). From the definition of restriction, we have

$$SE' = \{strid \mapsto SE(strid) \downarrow (L/strid) \mid strid \in \mathrm{Qual}(L)\} \tag{4.1}$$

$$TE' = TE \downarrow (L \cap \mathrm{TyCon}) \tag{4.2}$$

$$VE' = VE \downarrow (L \cap \mathrm{VId}) \tag{4.3}$$

By induction and from (4.1) and from the definition of restriction, we have

$$\begin{aligned} &\mathrm{Dom}\ SE \supseteq \mathrm{Dom}\ SE' \text{ and } SE(strid) \sqsupseteq SE'(strid) \\ &\quad \text{for all } strid \in \mathrm{Dom}\ SE' \end{aligned} \tag{4.4}$$

From (4.2) and from (4.3), we have

$$\begin{aligned} &\mathrm{Dom}\ TE \supseteq \mathrm{Dom}\ TE' \text{ and } TE(tycon) = TE'(tycon) \\ &\quad \text{for all } tycon \in \mathrm{Dom}\ TE' \end{aligned} \tag{4.5}$$

$$\begin{aligned} &\mathrm{Dom}\ VE \supseteq \mathrm{Dom}\ VE' \text{ and } VE(vid) = VE'(vid) \\ &\quad \text{for all } vid \in \mathrm{Dom}\ VE' \end{aligned} \tag{4.6}$$

Now, from the definition of strong enrichment and from (4.4), (4.5), and (4.6), we have $E \sqsupseteq E'$, as required. □

Strong enrichment is closed with respect to environment modification and restriction; if an environment E' strongly enriches another environment $(E \downarrow (L \setminus\setminus \mathrm{Dom}\ E_0))$, for some other environment E_0 and some set of long identifiers L, then $(E' + E_0)$ strongly enriches $(E + E_0) \downarrow L$, provided the restriction of $(E + E_0)$ to L is defined. This proposition is essential so as to demonstrate that elaboration dependence holds for program phrases that declare identifiers, locally.

Proposition 4.3.2 *If* $E' \sqsupseteq (E \downarrow (L \setminus\setminus \text{Dom } E_0))$ *and* $\text{LongIds}(E + E_0) \supseteq (L \setminus (\text{LongStrId} \cup \text{FunId}))$ *then* $(E' + E_0) \sqsupseteq ((E + E_0) \downarrow L)$.

PROOF The proof is a straightforward non-inductive proof. Write $E_1 = (E \downarrow (L \setminus\setminus \text{Dom } E_0))$, write E_1 in the form (SE_1, TE_1, VE_1), and write E' in the form (SE', TE', VE'). Further, write E in the form (SE, TE, VE) and write E_0 in the form (SE_0, TE_0, VE_0). From the definition of restriction, we have

$$SE_1 = \{strid \mapsto SE(strid) \downarrow ((L \setminus\setminus \text{Dom } SE_0)(strid)) \mid strid \in \text{Qual}(L \setminus\setminus \text{Dom } SE_0)\} \tag{4.7}$$

$$TE_1 = TE \downarrow ((L \setminus\setminus \text{Dom } TE_0) \cap \text{TyCon}) \tag{4.8}$$

$$VE_1 = VE \downarrow ((L \setminus\setminus \text{Dom } VE_0) \cap \text{VId}) \tag{4.9}$$

From assumptions and the definition of strong enrichment, we have

$$\text{Dom } SE' \supseteq \text{Dom } SE_1 \text{ and } SE'(strid) \sqsupseteq SE_1(strid) \text{ for all } strid \in \text{Dom } SE_1 \tag{4.10}$$

$$\text{Dom } TE' \supseteq \text{Dom } TE_1 \text{ and } TE'(tycon) = TE_1(tycon) \text{ for all } tycon \in \text{Dom } TE_1 \tag{4.11}$$

$$\text{Dom } VE' \supseteq \text{Dom } VE_1 \text{ and } VE'(vid) = VE_1(vid) \text{ for all } vid \in \text{Dom } VE_1 \tag{4.12}$$

From Proposition 4.2.1 and assumptions, we have there exists an environment $E'_1 = (E + E_0) \downarrow L$. Write E'_1 in the form (SE'_1, TE'_1, VE'_1). From the definition of restriction, we have

$$SE'_1 = \{strid \mapsto (SE + SE_0)(strid) \downarrow (L/strid) \mid strid \in \text{Qual}(L)\} \tag{4.13}$$

$$TE'_1 = (TE + TE_0) \downarrow (L \cap \text{TyCon}) \tag{4.14}$$

$$VE'_1 = (VE + VE_0) \downarrow (L \cap \text{VId}) \tag{4.15}$$

We have

$$((L \cap \text{TyCon}) \setminus \text{Dom } TE_0) \cup \text{Dom } TE_0 \supseteq L \cap \text{TyCon}$$

Thus, from the definition of curtailment and because $\text{Dom } TE'_1 = (L \cap \text{TyCon})$ follows from (4.14), we have

$$((L \setminus\setminus \text{Dom } TE_0) \cap \text{TyCon}) \cup \text{Dom } TE_0 \supseteq \text{Dom } TE'_1 \tag{4.16}$$

From (4.8) and from the definition of restriction, we have Dom $TE_1 = ((L \setminus\setminus \text{Dom } TE_0) \cap \text{TyCon})$, hence, from (4.16), we have

$$\text{Dom}(TE_1 + TE_0) \supseteq \text{Dom } TE'_1 \tag{4.17}$$

Similarly, we have

$$\text{Dom}(VE_1 + VE_0) \supseteq \text{Dom } VE'_1 \tag{4.18}$$

Now, assume $tycon \in \text{Dom } TE'_1$. From (4.14) and the definition of restriction, we have $TE'_1 = (TE \downarrow (L \cap \text{Dom } TE)) + (TE_0 \downarrow (L \cap \text{Dom } TE_0))$. It follows, that if $tycon \in (L \cap \text{Dom } TE_0)$ then $(TE' + TE_0)(tycon) = TE'_1(tycon)$. Otherwise, if $tycon \in ((L \cap \text{Dom } TE) \setminus \text{Dom } TE_0)$ then we have $TE'_1(tycon) = TE(tycon)$, so from (4.8) it follows that $TE'_1(tycon) = TE_1(tycon)$ and from (4.11) it follows that $TE'_1(tycon) = TE'(tycon)$. Thus, we have shown that

$$(TE' + TE_0)(tycon) = TE'_1(tycon) \text{ for all } tycon \in \text{Dom } TE'_1 \tag{4.19}$$

Similarly, we have

$$(VE' + VE_0)(vid) = VE'_1(vid) \text{ for all } vid \in \text{Dom } VE'_1 \tag{4.20}$$

From the definitions of qualifier and curtailment, we have $\text{Qual}(L \setminus\setminus \text{Dom } SE_0) \cup \text{Dom } SE_0 \supseteq \text{Qual}(L)$, thus, from (4.7) and (4.13), we have

$$\text{Dom}(SE_1 + SE_0) \supseteq \text{Dom } SE'_1 \tag{4.21}$$

Now, assume $strid \in \text{Dom } SE'_1$. From (4.13), we have

$$\begin{aligned} SE'_1 = \{strid \mapsto SE(strid) \downarrow (L/strid) \mid strid \in \text{Qual}(L) \setminus \text{Dom } SE_0\} + \\ \{strid \mapsto SE_0(strid) \downarrow (L/strid) \mid strid \in \text{Qual}(L) \cap \text{Dom } SE_0\} \end{aligned} \tag{4.22}$$

It follows that if $strid \in \text{Qual}(L) \cap \text{Dom } SE_0$ then we have from (4.22) that there exists an environment E'_0 such that $E'_0 = SE_0(strid) \downarrow (L/strid)$. It now follows from Proposition 4.3.1 that $SE_0(strid) \sqsupseteq (SE_0 \downarrow (L/strid))$ and because $SE_0(strid) = (SE' + SE_0)(strid)$ it follows from (4.22) that we have $(SE' + SE_0)(strid) \sqsupseteq SE'_1(strid)$. Otherwise, if $strid \in \text{Qual}(L) \setminus \text{Dom } SE_0$ then we have from (4.10) that $SE'(strid) \sqsupseteq SE_1(strid)$. Moreover, we have $(SE' + SE_0)(strid) = SE'(strid)$ and from (4.7) and the definition

of curtailment, we have $SE_1(strid) = SE(strid) \downarrow (L/strid)$. It follows that we have $(SE' + SE_0)(strid) \sqsupseteq SE'_1(strid)$, thus, we can conclude

$$(SE' + SE_0)(strid) \sqsupseteq SE'_1(strid) \text{ for all } strid \in \text{Dom } SE'_1 \tag{4.23}$$

Now, from the definition of strong enrichment and from (4.17), (4.18), (4.19), (4.20), (4.21), and (4.23), we have $(E' + E_0) \sqsupseteq E'_1$, as required. □

The preceding proposition extends to bases as follows:

Proposition 4.3.3 *If* $B' \sqsupseteq (B \downarrow (L \setminus\setminus \text{Dom } B_0))$ *and* $\text{LongIds}(B + B_0) \supseteq (L \setminus \text{LongStrId})$ *then* $(B' + B_0) \sqsupseteq ((B + B_0) \downarrow L)$.

PROOF Write B' on the form (F', E'), write B on the form (F, E), and write B_0 on the form (F_0, E_0). Further, let $L' = (L \setminus \text{Dom } F_0) \cap \text{FunId}$. From assumptions and from the definitions of restriction and strong enrichment, we have

$$\begin{aligned} &\text{Dom } F' \supseteq L' \text{ and } F'(funid) = (F \downarrow L')(funid) \\ &\quad \text{for all } funid \in L' \end{aligned} \tag{4.24}$$

$$E' \sqsupseteq (E \downarrow (L \setminus\setminus \text{Dom } E_0)) \tag{4.25}$$

From assumptions, we have $\text{LongIds}(F + F_0) \cup \text{LongIds}(E + E_0) \supseteq (L \setminus \text{LongStrId})$, thus, from the definition of LongIds, we have

$$\text{Dom}(F + F_0) \supseteq (L \cap \text{FunId}) \tag{4.26}$$

$$\text{LongIds}(E + E_0) \supseteq (L \setminus (\text{LongStrId} \cup \text{FunId})) \tag{4.27}$$

From (4.24) and from (4.26), we have

$$\begin{aligned} &\text{Dom}(F' + F_0) \supseteq (L \cap \text{FunId}) \text{ and} \\ &\quad (F' + F_0)(funid) = ((F + F_0) \downarrow (L \cap \text{FunId}))(funid) \\ &\quad \text{for all } funid \in (L \cap \text{FunId}) \end{aligned} \tag{4.28}$$

Moreover, from Proposition 4.3.2, from (4.25), and from (4.27), we have

$$(E' + E_0) \sqsupseteq ((E + E_0) \downarrow L) \tag{4.29}$$

Now, from (4.28) and (4.29) and from the definitions of restriction and strong enrichment, we have $(B' + B_0) \sqsupseteq ((B + B_0) \downarrow L)$, as required. □

The following proposition is essential for demonstrating elaboration dependence for ModML program phrases that contain sub-phrases.

Proposition 4.3.4 *If* $E' \sqsupseteq (E \downarrow L)$ *and* $L' \subseteq L$ *then* $E' \sqsupseteq (E \downarrow L')$.

PROOF The proof is by induction on the structure of E. Let $E_0 = E \downarrow L$. Write E' in the form (SE', TE', VE'), write E in the form (SE, TE, VE), and write E_0 in the form (SE_0, TE_0, VE_0). From the definition of restriction, we have

$$SE_0 = \{strid \mapsto SE(strid) \downarrow (L/strid) \mid strid \in \mathrm{Qual}(L)\} \quad (4.30)$$

$$TE_0 = TE \downarrow (L \cap \mathrm{TyCon}) \quad (4.31)$$

$$VE_0 = VE \downarrow (L \cap \mathrm{VId}) \quad (4.32)$$

Moreover, from the definition of strong enrichment, we have

$$\mathrm{Dom}\ SE' \supseteq \mathrm{Dom}\ SE_0 \text{ and } SE'(strid) \sqsupseteq SE_0(strid) \text{ for all } strid \in \mathrm{Dom}\ SE_0 \quad (4.33)$$

$$\mathrm{Dom}\ TE' \supseteq \mathrm{Dom}\ TE_0 \text{ and } TE'(tycon) = TE_0(tycon) \text{ for all } tycon \in \mathrm{Dom}\ TE_0 \quad (4.34)$$

$$\mathrm{Dom}\ VE' \supseteq \mathrm{Dom}\ VE_0 \text{ and } VE'(vid) = VE_0(vid) \text{ for all } vid \in \mathrm{Dom}\ VE_0 \quad (4.35)$$

Now, let $E'' = E \downarrow L'$ and write E'' in the form (SE'', TE'', VE''). From the definition of restriction, we have

$$SE'' = \{strid \mapsto SE(strid) \downarrow (L'/strid) \mid strid \in \mathrm{Qual}(L')\} \quad (4.36)$$

$$TE'' = TE \downarrow (L' \cap \mathrm{TyCon}) \quad (4.37)$$

$$VE'' = VE \downarrow (L' \cap \mathrm{VId}) \quad (4.38)$$

From assumptions and from Proposition 4.1.2, we have

$$(L'/strid) \subseteq (L/strid) \text{ for all } strid \in \mathrm{Qual}(L') \quad (4.39)$$

Moreover, from (4.30), from (4.32), and from Proposition 4.1.1, we have

$$SE'(strid) \sqsupseteq (SE(strid) \downarrow (L/strid) \text{ for all } strid \in \mathrm{Qual}(L') \quad (4.40)$$

From (4.39) and (4.40), we can now apply induction for each $strid \in \mathrm{Qual}(L')$ to get

$$SE'(strid) \sqsupseteq (SE(strid) \downarrow (L'/strid) \text{ for all } strid \in \mathrm{Qual}(L') \quad (4.41)$$

Thus, from (4.30), (4.36), and (4.41), we have

$$\begin{aligned}&\text{Dom } SE' \supseteq \text{Dom } SE'' \text{ and } SE'(strid) \sqsupseteq SE''(strid)\\ &\quad \text{for all } strid \in \text{Dom } SE''\end{aligned} \tag{4.42}$$

Also, because Dom $TE'' \subseteq$ Dom TE_0 and Dom $VE'' \subseteq$ Dom VE_0, it follows from (4.31), (4.32), (4.34), (4.35), (4.37), and (4.38) that

$$\begin{aligned}&\text{Dom } TE' \supseteq \text{Dom } TE'' \text{ and } TE'(tycon) = TE''(tycon)\\ &\quad \text{for all } tycon \in \text{Dom } TE''\end{aligned} \tag{4.43}$$

$$\begin{aligned}&\text{Dom } VE' \supseteq \text{Dom } VE'' \text{ and } VE'(vid) = VE''(vid)\\ &\quad \text{for all } vid \in \text{Dom } VE''\end{aligned} \tag{4.44}$$

Now, from the definition of strong enrichment and from (4.42), (4.43), and (4.44), we have $E' \sqsupseteq E''$, as required. □

The preceding proposition extends to bases:

Proposition 4.3.5 *If $B' \sqsupseteq (B \downarrow L)$ and $L' \subseteq L$ then $B' \sqsupseteq (B \downarrow L')$.*

PROOF Write B' in the form (F', E') and write B in the form (F, E). From the definitions of restriction and strong enrichment, we have

$$\begin{aligned}&\text{Dom } F' \supseteq \text{Dom}(F \downarrow (L \cap \text{FunId})) \text{ and}\\ &\quad F'(funid) = (F \downarrow (L \cap \text{FunId}))(funid)\\ &\quad \text{for all } funid \in (L \cap \text{FunId})\end{aligned} \tag{4.45}$$

$$E' \sqsupseteq (E \downarrow L) \tag{4.46}$$

From assumptions and from (4.45), we have

$$\begin{aligned}&\text{Dom } F' \supseteq (L' \cap \text{FunId}) \text{ and } F'(funid) = (F \downarrow (L' \cap \text{FunId}))(funid)\\ &\quad \text{for all } funid \in (L' \cap \text{FunId})\end{aligned} \tag{4.47}$$

Moreover, from (4.46) and from Proposition 4.3.4, we have

$$E' \sqsupseteq (E \downarrow L') \tag{4.48}$$

Now, from the definitions of restriction and strong enrichment and from (4.47) and (4.48), we have $B' \sqsupseteq (B \downarrow L')$, as required. □

The following proposition is essential for demonstrating elaboration dependence for ModML program phrases that refer to assumptions for long type constructors or for long value identifiers.

Proposition 4.3.6 *If* $E' \sqsupseteq (E \downarrow L)$ *and* $longx \in (L \cap (\text{LongTyCon} \cup \text{LongVId}))$ *then* $E'(longx) = E(longx)$.

PROOF The proof is by induction over the structure of $longx$. There are two cases.

If $longx \in (\text{TyCon} \cup \text{VId})$, it follows immediately from definitions of restriction and strong enrichment that $E'(longx) = E(longx)$, as required.

Otherwise, there exist a structure identifier $strid$ and a long identifier $longx'$ such that $longx = strid.longx'$. From the definitions of restriction and strong enrichment, we have $E'(strid) \sqsupseteq (E(strid) \downarrow (L/strid))$. Moreover, from assumptions and from the definition of projection, we have $longx' \in ((L/strid) \cap (\text{LongTyCon} \cup \text{LongVId}))$. It follows that we can apply induction to get $(E'(strid))(longx') = (E(strid))(longx')$, hence, we have $E'(longx) = E(longx)$, as required. □

The preceding proposition extends to bases:

Proposition 4.3.7 *If* $B' \sqsupseteq (B \downarrow L)$ *and* $longx \in (L \cap (\text{FunId} \cup \text{LongTyCon} \cup \text{LongVId}))$ *then* $B'(longx) = B(longx)$.

PROOF There are two cases.

If $longx \in \text{FunId}$, it follows immediately from the definitions of restriction and strong enrichment that $B'(longx) = B(longx)$, as required.

Otherwise, we have $longx \in (\text{LongTyCon} \cap \text{LongVId})$. From the definitions of restriction and strong enrichment and from Proposition 4.3.6, we have $B'(longx) = B(longx)$, as required. □

4.4 Agreement

The set of *value constructors* of an environment E, written $\text{Cons}(E)$, is defined as

$$\text{Cons}(E) = \{vid \mid \exists \sigma \text{ s.t. } E(vid) = (\sigma, \mathsf{c})\}$$

An environment E_1 *agrees* with another environment E_2 w.r.t. a set of long identifiers L, written $E_1 \approx_L E_2$, iff $E_1(longstrid) = E_2(longstrid)$ for all $longstrid \in L$ and $\text{Cons}(E_1) = \text{Cons}(E_2)$. Further, a basis $B_1 = (F_1, E_1)$ *agrees* with another basis $B_2 = (F_2, E_2)$ w.r.t. a set of long identifiers L, written $B_1 \approx_L B_2$, iff $E_1 \approx_L E_2$.

4.5 Core Dependence

The functions for finding free and declared identifiers are defined by mutual recursion. The notations $\mathrm{fid}(\mathit{phrase}) = L$ and $\mathrm{decl}(\mathit{phrase}) = I$ are overloaded and used for all phrases *phrase*. For the Core language, only declarations declare identifiers, hence, we define $\mathrm{decl}(\mathit{phrase}) = I$ only when *phrase* is a declaration.

Type Expressions $\boxed{\mathrm{fid}(ty) = L}$

$$\begin{aligned}
\mathrm{fid}(ty_1 \texttt{ -> } ty_2) &= \mathrm{fid}(ty_1) \cup \mathrm{fid}(ty_2) && (4.49)\\
\mathrm{fid}(\mathit{tyvar}) &= \emptyset && (4.50)\\
\mathrm{fid}(\mathit{tyseq}\ \mathit{longtycon}) &= \{\mathit{longtycon}\} \cup && (4.51)\\
&\quad \mathrm{fid}(ty_1) \cup \cdots \cup \mathrm{fid}(ty_k) &&\\
\text{where } \mathit{tyseq} = ty_1 \cdots ty_k & && (4.52)
\end{aligned}$$

Proposition 4.5.1 *If* $E \vdash ty \Rightarrow \tau$ *then* $\mathrm{LongIds}(E) \supseteq \mathrm{fid}(ty)$.

PROOF By induction over the structure of ty. □

The following proposition states that elaboration of a type expression depends only on assumptions for those long identifiers that occur free in the type expression.

Proposition 4.5.2 (Type expression dependence) *If* $E \vdash ty \Rightarrow \tau$ *and* $E' \sqsupseteq (E \downarrow \mathrm{fid}(ty))$ *then* $E' \vdash ty \Rightarrow \tau$.

PROOF The proof is by induction over the structure of ty and proceeds by case analysis.

$\boxed{\text{CASE } ty = ty_1 \texttt{ -> } ty_2}$ From assumptions and from rule 2.1, we have

$$E \vdash ty_1 \Rightarrow \tau_1 \quad (4.53)$$
$$E \vdash ty_2 \Rightarrow \tau_2 \quad (4.54)$$
$$\tau = \tau_1 \to \tau_2 \quad (4.55)$$

From assumptions, from (4.49), and from Proposition 4.3.4, we have

$$E' \sqsupseteq (E \downarrow \mathrm{fid}(ty_1)) \quad (4.56)$$
$$E' \sqsupseteq (E \downarrow \mathrm{fid}(ty_2)) \quad (4.57)$$

We can now apply induction twice to (4.53) and (4.56), and to (4.54) and (4.57) to get

$$E' \vdash ty_1 \Rightarrow \tau_1 \tag{4.58}$$
$$E' \vdash ty_2 \Rightarrow \tau_2 \tag{4.59}$$

From rule 2.1 and from (4.58), (4.59), and (4.55), we have $E' \vdash ty \Rightarrow \tau$, as required.

CASE $ty = tyvar$ From assumptions and from rule 2.2, we have $E' \vdash tyvar \Rightarrow \alpha$, as required.

CASE $ty = tyseq\ longtycon$ Write *tyseq* in the form $ty_1 \cdots ty_k$. From assumptions and from rule 2.3, we have

$$E(longtycon) = (\theta^k, VE) \tag{4.60}$$
$$E \vdash ty_i \Rightarrow \tau_i, \quad i = 1..k \tag{4.61}$$
$$\tau = (\tau_1, \cdots, \tau_k)\theta^k \tag{4.62}$$

From assumptions, from (4.60), from (4.51), and from Proposition 4.3.6, we have

$$E'(longtycon) = (\theta^k, VE) \tag{4.63}$$

From assumptions, from (4.51), and from Proposition 4.3.4, we have

$$E' \sqsupseteq (E \downarrow \mathrm{fid}(ty_i)), \quad i = 1..k \tag{4.64}$$

We can now apply induction k times to (4.61) and (4.64) to get

$$E' \vdash ty_i \Rightarrow \tau_i, \quad i = 1..k \tag{4.65}$$

From (4.65), from (4.63), from (4.62) and from rule 2.3, we have $E' \vdash ty \Rightarrow \tau$, as required. □

Expressions $\boxed{\mathrm{fid}(exp) = L}$

$$\mathrm{fid}(longvid) = \{longvid\} \tag{4.66}$$
$$\mathrm{fid}(\texttt{fn}^{\texttt{v}}\ vid\ \texttt{=>}\ exp) = \mathrm{fid}(exp) \setminus \{vid\} \tag{4.67}$$
$$\mathrm{fid}(\texttt{fn}^{\texttt{c}}\ longvid\ \texttt{=>}\ exp) = \mathrm{fid}(exp) \cup \{longvid\} \tag{4.68}$$
$$\mathrm{fid}(exp_1\ exp_2) = \mathrm{fid}(exp_1) \cup \mathrm{fid}(exp_2) \tag{4.69}$$
$$\mathrm{fid}(\texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}) = \mathrm{fid}(dec) \cup (\mathrm{fid}(exp) \setminus\!\setminus \mathrm{decl}(dec)) \tag{4.70}$$

Declarations $\boxed{\mathrm{fid}(dec) = L}$

$$\begin{aligned}
\mathrm{fid}(\texttt{open}^I\ longstrid) &= \{longstrid\} && (4.71)\\
\mathrm{fid}(\texttt{val}\ vid\ \texttt{=}\ exp) &= \mathrm{fid}(exp) && (4.72)\\
\mathrm{fid}(\texttt{type}\ tyvarseq\ tycon\ \texttt{=}\ ty) &= \mathrm{fid}(ty) && (4.73)\\
\mathrm{fid}(\texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid) &= \emptyset && (4.74)
\end{aligned}$$

$\boxed{\mathrm{decl}(dec) = I}$

$$\begin{aligned}
\mathrm{decl}(\texttt{open}^I\ longstrid) &= I && (4.75)\\
\mathrm{decl}(\texttt{val}\ vid\ \texttt{=}\ exp) &= \{vid\} && (4.76)\\
\mathrm{decl}(\texttt{type}\ tyvarseq\ tycon\ \texttt{=}\ ty) &= \{tycon\} && (4.77)\\
\mathrm{decl}(\texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid) &= \{tycon, vid\} && (4.78)
\end{aligned}$$

Proposition 4.5.3 *If* $E \vdash dec \Rightarrow (T)E'$ *then* $\mathrm{LongIds}(E) \supseteq (\mathrm{fid}(dec) \setminus \mathrm{LongStrId})$ *and* $\mathrm{decl}(dec) = \mathrm{Dom}\, E'$. *Also, if* $E \vdash exp \Rightarrow \tau$ *then* $\mathrm{LongIds}(E) \supseteq (\mathrm{fid}(exp) \setminus \mathrm{LongStrId})$.

PROOF By induction over the structure of *dec* and *exp*. □

The following proposition states that elaboration of expressions and declarations depends on assumptions for those long identifiers that occur free in the phrase. Agreement is used to express that elaboration depends on the assumptions for free long structure identifiers of the phrase and on the set of value identifiers with constructor status in the assumptions.

Proposition 4.5.4 (Core dependence) *Let phrase be either an expression or a declaration and let A be either a type or a signature. If* $E \vdash phrase \Rightarrow A$ *and* $E' \sqsupseteq (E \downarrow \mathrm{fid}(phrase))$ *and* $E' \approx_{\mathrm{fid}(phrase)} E$ *then* $E' \vdash phrase \Rightarrow A$.

PROOF The proof is by induction over the structure of *phrase*.

$\boxed{\text{CASE } exp = longvid}$ From assumptions and from rule 2.4, we have $E(longvid) = (\sigma, is)$ and $\sigma \succ \tau$, hence, from Proposition 4.3.6, from assumptions, and because $\mathrm{fid}(longvid) = \{longvid\}$, we have $E'(longvid) = (\sigma, is)$. Thus, we have from rule 2.4 that $E' \vdash longvid \Rightarrow \tau$, as required.

CASE $exp = \texttt{fn}^{\texttt{v}}\ vid\ \texttt{=>}\ exp'$ From assumptions and from rule 2.5, we have

$$vid \notin \text{Dom } E \text{ or } is \text{ of } E(vid) = \texttt{v} \tag{4.79}$$
$$E + E_0 \vdash exp' \Rightarrow \tau' \tag{4.80}$$
$$E \vdash exp \Rightarrow \tau \to \tau' \tag{4.81}$$
$$E_0 = \{vid \mapsto (\tau, \texttt{v})\} \tag{4.82}$$

From the definition of agreement and from (4.79), we have

$$vid \notin \text{Dom } E' \text{ or } is \text{ of } E'(vid) = \texttt{v} \tag{4.83}$$

From (4.67), from the definition of curtailment, and from (4.82), we have

$$\text{fid}(exp) = \text{fid}(exp') \setminus\setminus \text{Dom } E_0 \tag{4.84}$$

Now, from Proposition 4.5.3 and from (4.80), we have

$$\text{LongIds}(E + E_0) \supseteq (\text{fid}(exp') \setminus (\text{LongStrId} \cup \text{FunId})) \tag{4.85}$$

From Proposition 4.3.2, from assumptions, and from (4.84) and (4.85), we have

$$(E' + E_0) \sqsupseteq ((E + E_0) \downarrow \text{fid}(exp')) \tag{4.86}$$

Moreover, from assumptions and from the definition of agreement, we have

$$(E' + E_0) \approx_{\text{fid}(exp')} (E + E_0) \tag{4.87}$$

By applying induction to (4.80), (4.86), and (4.87), we have

$$E' + E_0 \vdash exp' \Rightarrow \tau' \tag{4.88}$$

Now, from rule 2.5 and from (4.83), (4.82), and (4.88), we have $E' \vdash exp \Rightarrow \tau \to \tau'$, as required.

CASE $exp = \texttt{let}\ dec\ \texttt{in}\ exp'\ \texttt{end}$ From assumptions and from rule 2.8, we have

$$E \vdash dec \Rightarrow (T)E'' \tag{4.89}$$
$$E + E'' \vdash exp' \Rightarrow \tau \tag{4.90}$$
$$T \cap (\text{tynames}(E, \tau)) = \emptyset \tag{4.91}$$

By renaming of bound names of $(T)E''$, we can assume

$$T \cap (\text{tynames}(E', \tau)) = \emptyset \tag{4.92}$$

From (4.70) and from Proposition 4.5.3, we have

$$\text{fid}(exp) = \text{fid}(dec) \cup (\text{fid}(exp') \setminus\setminus \text{Dom } E'') \tag{4.93}$$

Now, from assumptions and from (4.93) and from Proposition 4.3.4, we have

$$E' \sqsupseteq (E \downarrow \text{fid}(dec)) \tag{4.94}$$

Moreover, from assumptions and from the definition of agreement, we have

$$E' \approx_{\text{fid}(dec)} E \tag{4.95}$$

We can now apply induction to (4.89), (4.94), and (4.95) to get

$$E' \vdash dec \Rightarrow (T)E'' \tag{4.96}$$

From assumptions, from (4.93), and from Proposition 4.3.4, we have

$$E' \sqsupseteq (E \downarrow (\text{fid}(exp') \setminus\setminus \text{Dom } E'')) \tag{4.97}$$

Moreover, from (4.90) and from Proposition 4.5.3, we have

$$\text{LongIds}(E + E'') \supseteq (\text{fid}(exp') \setminus \text{LongStrId}) \tag{4.98}$$

Now, from Proposition 4.3.2, from (4.97) and (4.98), and because no functor identifiers occur free in Core-level expressions, we have

$$(E' + E'') \sqsupseteq ((E + E'') \downarrow \text{fid}(exp')) \tag{4.99}$$

Further, it follows from assumptions, from (4.93), and from the definition of agreement that

$$(E' + E'') \approx_{\text{fid}(exp')} (E + E'') \tag{4.100}$$

We can now apply induction to (4.90), (4.99), and (4.100) to get

$$E' + E'' \vdash exp' \Rightarrow \tau \tag{4.101}$$

From rule 2.8 and from (4.96), (4.92), and (4.101), we have $E' \vdash exp \Rightarrow \tau$, as required.

CASE $dec = \texttt{val}\ vid\ \texttt{=}\ exp$ From assumptions and from rule 2.9, we have

$$E \vdash exp \Rightarrow \tau \tag{4.102}$$

$$\text{tyvars}\ \alpha^{(k)} \cap \text{tyvars}\ E = \emptyset \tag{4.103}$$

$$\Sigma = (\emptyset)\{vid \mapsto (\forall \alpha^{(k)}.\tau, \mathtt{v})\} \tag{4.104}$$

By renaming of bound names of $\forall \alpha^{(k)}.\tau$, we can assume

$$\text{tyvars}\ \alpha^{(k)} \cap \text{tyvars}\ E' = \emptyset \tag{4.105}$$

From (4.72), we have $\text{fid}(dec) = \text{fid}(exp)$, hence we can apply induction to (4.102) to get

$$E' \vdash exp \Rightarrow \tau \tag{4.106}$$

It now follows from rule 2.9 and from (4.105), (4.104), and (4.106) that $E' \vdash dec \Rightarrow \Sigma$, as required.

CASE $dec = \texttt{open}^I\ longstrid$ From assumptions and from rule 2.12, we have $E(longstrid) = E''$ and $\text{Dom}\ E'' = I$ and $E \vdash dec \Rightarrow (\emptyset)E''$. From assumptions, from (4.71), and from the definition of agreement, we have $E'(longstrid) = E''$, thus, from rule 2.12, we have $E' \vdash dec \Rightarrow (\emptyset)E''$, as required.

The proofs for the remaining cases are similar. □

4.6 Signature Dependence

Also specifications declare identifiers, hence, we also define $\text{decl}(spec) = I$.

Specifications $\boxed{\text{fid}(spec) = L}$

$$\text{fid}(\texttt{val}\ vid\ \texttt{:}\ ty) = \text{fid}(ty) \tag{4.107}$$

$$\text{fid}(\texttt{type}\ tyvarseq\ tycon) = \emptyset \tag{4.108}$$

$$\text{fid}(\texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid) = \emptyset \tag{4.109}$$

$$\text{fid}(spec_1\ spec_2) = \text{fid}(spec_1) \cup (\text{fid}(spec_2) \setminus\setminus \text{decl}(spec_1)) \tag{4.110}$$

$$\text{fid}(\varepsilon) = \emptyset \tag{4.111}$$

$\boxed{\text{decl}(spec) = I}$

$$\begin{aligned}
\text{decl}(\texttt{val}\ vid\ \texttt{:}\ ty) &= \{vid\} && (4.112)\\
\text{decl}(\texttt{type}\ tyvarseq\ tycon) &= \{tycon\} && (4.113)\\
\text{decl}(\texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid) &= \{tycon, vid\} && (4.114)\\
\text{decl}(spec_1\ spec_2) &= \text{decl}(spec_1) \cup && \\
&\quad\ \text{decl}(spec_2) && (4.115)\\
\text{decl}(\varepsilon) &= \emptyset && (4.116)
\end{aligned}$$

Signature Expressions $\boxed{\text{fid}(sigexp) = L}$

$$\begin{aligned}
\text{fid}(\texttt{sig}\ spec\ \texttt{end}) &= \text{fid}(spec) && (4.117)\\
\text{fid}(sigexp\ \texttt{where type}\ tyvarseq\ \texttt{=}\ ty) &= \text{fid}(sigexp) \cup \text{fid}(ty) && (4.118)
\end{aligned}$$

Proposition 4.6.1 *If* $B \vdash spec \Rightarrow (T)E$ *then* $\text{LongIds}(B) \supseteq \text{fid}(spec)$ *and* $\text{decl}(spec) = \text{Dom}\ E$. *Also, if* $B \vdash sigexp \Rightarrow (T)E$ *then* $\text{LongIds}(B) \supseteq \text{fid}(sigexp)$.

PROOF By induction over the structure of *spec* and *sigexp*. □

Elaboration of signature expressions and specifications depends on assumptions for those long identifiers occurring free in the phrase.

Proposition 4.6.2 (Signature dependence) *Let phrase be either a specification or a signature expression. If* $B \vdash phrase \Rightarrow \Sigma$ *and* $B' \sqsupseteq (B \downarrow \text{fid}(phrase))$ *then* $B' \vdash phrase \Rightarrow \Sigma$.

PROOF The proof is by induction over the structure of *phrase*. We show two of the cases.

$\boxed{\text{CASE } spec = spec_1\ spec_2}$ From assumptions and from rule 2.17, we have

$$\begin{aligned}
&\text{Dom}\ E_1 \cap \text{Dom}\ E_2 = \emptyset && (4.119)\\
&B \vdash spec_1 \Rightarrow (T_1)E_1 && (4.120)\\
&(T_1 \cup T_2) \cap \text{tynames}\ B = \emptyset && (4.121)\\
&B + E_1 \vdash spec_2 \Rightarrow (T_2)E_2 && (4.122)\\
&T_2 \cap (T_1 \cup \text{tynames}\ E_1) = \emptyset && (4.123)\\
&\Sigma = (T_1 \cup T_2)(E_1 + E_2) && (4.124)
\end{aligned}$$

From assumptions and from (4.110), we have

$$B' \sqsupseteq (B \downarrow (\mathrm{fid}(spec_1) \cup (\mathrm{fid}(spec_2) \setminus\setminus \mathrm{decl}(spec_1)))) \tag{4.125}$$

Now, from Proposition 4.3.5 and from (4.125), we have

$$B' \sqsupseteq (B \downarrow \mathrm{fid}(spec_1)) \tag{4.126}$$
$$B' \sqsupseteq (B \downarrow (\mathrm{fid}(spec_2) \setminus\setminus \mathrm{decl}(spec_1))) \tag{4.127}$$

Thus, we can now apply induction to (4.120) and (4.126) to get

$$B' \vdash spec_1 \Rightarrow (T_1)E_1 \tag{4.128}$$

From Proposition 4.6.1 and from (4.120), we have $\mathrm{decl}(spec_1) = \mathrm{Dom}\ E_1$, hence from (4.127), we have

$$B' \sqsupseteq (B \downarrow (\mathrm{fid}(spec_1) \setminus\setminus \mathrm{Dom}\ E_1)) \tag{4.129}$$

Now, from Proposition 4.6.1 and from (4.122) and because no long structure identifiers occur free in specifications, we have

$$\mathrm{LongIds}(B + E_1) \supseteq (\mathrm{fid}(spec_2) \setminus \mathrm{LongStrId}) \tag{4.130}$$

Moreover, from Proposition 4.3.3, from (4.129), and from (4.130), we have

$$(B' + E_1) \sqsupseteq ((B + E_1) \downarrow \mathrm{fid}(spec_2)) \tag{4.131}$$

We can now apply induction to (4.122) and (4.131) to get

$$B' + E_1 \vdash spec_2 \Rightarrow (T_2)E_2 \tag{4.132}$$

By appropriate renaming of bound names of Σ, we can assume

$$(T_1 \cup T_2) \cap \mathrm{tynames}\ B' = \emptyset \tag{4.133}$$

From rule 2.17 and from (4.119), (4.128), (4.133), (4.132), (4.123), and (4.124), we have $B' \vdash spec \Rightarrow \Sigma$, as required.

CASE $sigexp = sigexp'$ `where type` $tyvarseq\ longtycon$ `=` ty From assumptions and from rule 2.20, we have

$$B \vdash sigexp' \Rightarrow (T)E \tag{4.134}$$

$$T \cap \text{tynames } B = \emptyset \tag{4.135}$$
$$tyvarseq = \alpha^{(k)} \tag{4.136}$$
$$E \text{ of } B \vdash ty \Rightarrow \tau \tag{4.137}$$
$$E(longtycon) = (t, VE) \tag{4.138}$$
$$t \in T \tag{4.139}$$
$$\varphi = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \tag{4.140}$$
$$\Sigma = (T)(\varphi(E)) \tag{4.141}$$

From assumptions and from (4.118), we have

$$B' \sqsupseteq (B \downarrow (\text{fid}(sigexp') \cup \text{fid}(ty))) \tag{4.142}$$

It follows from Proposition 4.3.5 and from (4.142) that we have

$$B' \sqsupseteq (B \downarrow \text{fid}(sigexp')) \tag{4.143}$$
$$B' \sqsupseteq (B \downarrow \text{fid}(ty)) \tag{4.144}$$

We can now apply induction to (4.134) and (4.143) to get

$$B' \vdash sigexp' \Rightarrow (T)E \tag{4.145}$$

Moreover, from the definitions of restriction and strong enrichment and from (4.144), we have

$$(E \text{ of } B') \sqsupseteq ((E \text{ of } B) \downarrow \text{fid}(ty)) \tag{4.146}$$

Thus, from Proposition 4.5.2 and from (4.146), we have

$$E \text{ of } B' \vdash ty \Rightarrow \tau \tag{4.147}$$

By appropriate renaming of bound names of Σ, we can assume

$$T \cap \text{tynames } B' = \emptyset \tag{4.148}$$

We now have from rule 2.20 and from (4.145), (4.148), (4.136), (4.147), (4.138), (4.139), (4.140), and (4.141) that $B' \vdash sigexp \Rightarrow \Sigma$, as required.

The proofs for the remaining cases are similar. □

4.7 Module Dependence

Both structure-level declarations and top-level declarations may declare identifiers, hence, we define $\mathrm{decl}(\mathit{phrase}) = I$ for both of these phrase classes.

Structure-level Expressions $\boxed{\mathrm{fid}(\mathit{strexp}) = L}$

$$\begin{aligned}
\mathrm{fid}(\texttt{struct}\ \mathit{strdec}\ \texttt{end}) &= \mathrm{fid}(\mathit{strdec}) && (4.149)\\
\mathrm{fid}(\mathit{longstrid}) &= \{\mathit{longstrid}\} && (4.150)\\
\mathrm{fid}(\mathit{strexp}\ \texttt{:}\ \mathit{sigexp}) &= \mathrm{fid}(\mathit{strexp}) \cup \mathrm{fid}(\mathit{sigexp}) && (4.151)\\
\mathrm{fid}(\mathit{strexp}\ \texttt{:>}\ \mathit{sigexp}) &= \mathrm{fid}(\mathit{strexp}) \cup \mathrm{fid}(\mathit{sigexp}) && (4.152)\\
\mathrm{fid}(\mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)}) &= \{\mathit{funid}\} \cup \mathrm{fid}(\mathit{strexp}) && (4.153)
\end{aligned}$$

Structure-level Declarations $\boxed{\mathrm{fid}(\mathit{strdec}) = L}$

$$\begin{aligned}
\mathrm{fid}(\mathit{dec}) &= \mathrm{fid}(\mathit{dec}) && (4.154)\\
\mathrm{fid}(\texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp}) &= \mathrm{fid}(\mathit{strexp}) && (4.155)\\
\mathrm{fid}(\mathit{strdec}_1\ \mathit{strdec}_2) &= \mathrm{fid}(\mathit{strdec}_1) \cup && (4.156)\\
&\quad (\mathrm{fid}(\mathit{strdec}_2) \setminus\setminus \mathrm{decl}(\mathit{strdec}_1))\\
\mathrm{fid}(\varepsilon) &= \emptyset && (4.157)
\end{aligned}$$

$\boxed{\mathrm{decl}(\mathit{strdec}) = I}$

$$\begin{aligned}
\mathrm{decl}(\mathit{dec}) &= \mathrm{decl}(\mathit{dec}) && (4.158)\\
\mathrm{decl}(\texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp}) &= \{\mathit{strid}\} && (4.159)\\
\mathrm{decl}(\mathit{strdec}_1\ \mathit{strdec}_2) &= \mathrm{decl}(\mathit{strdec}_1) \cup\\
&\quad \mathrm{decl}(\mathit{strdec}_2) && (4.160)\\
\mathrm{decl}(\varepsilon) &= \emptyset && (4.161)
\end{aligned}$$

Proposition 4.7.1 *If* $B \vdash \mathit{strdec} \Rightarrow (T)E$ *then* $\mathrm{LongIds}(B) \supseteq (\mathrm{fid}(\mathit{strdec}) \cap \mathrm{LongStrId})$ *and* $\mathrm{decl}(\mathit{strdec}) = \mathrm{Dom}\ E$. *Also, if* $B \vdash \mathit{strexp} \Rightarrow (T)E$ *then* $\mathrm{LongIds}(B) \supseteq (\mathrm{fid}(\mathit{strexp}) \cap \mathrm{LongStrId})$.

PROOF By induction over the structure of *strdec* and *strexp*. □

The following proposition states that elaboration of structure-level expressions and structure-level declarations depends on assumptions for those long identifiers occurring free in the phrase.

Proposition 4.7.2 (Module dependence) *Let phrase be either a structure declaration or a structure expression. If* $B \vdash phrase \Rightarrow \Sigma$ *and* $B' \sqsupseteq (B \downarrow \mathrm{fid}(phrase))$ *and* $B' \approx_{\mathrm{fid}(phrase)} B$ *then* $B' \vdash phrase \Rightarrow \Sigma$.

PROOF The proof is by induction over the structure of *phrase*.

CASE $strexp = longstrid$ From assumptions and from rule 2.24, we have

$$B(longstrid) = E \tag{4.162}$$

From assumptions and from (4.150), we have $B' \approx_{\{longstrid\}} B$, hence, from the definition of agreement and from (4.162), we have $B'(longstrid) = E$. It follows from rule 2.24 that we have $B' \vdash longstrid \Rightarrow (\emptyset)E$, as required.

CASE $strexp = strexp' : sigexp$ From assumptions and from rule 2.25, we have

$$B \vdash strexp \Rightarrow (T)E \tag{4.163}$$
$$B \vdash sigexp \Rightarrow \Sigma' \tag{4.164}$$
$$\Sigma' \geq E' \prec E \tag{4.165}$$
$$T \cap \mathrm{tynames}\ B = \emptyset \tag{4.166}$$
$$\Sigma = (T)E' \tag{4.167}$$

By appropriate renaming of bound names of Σ, we can assume

$$T \cap \mathrm{tynames}\ B' = \emptyset \tag{4.168}$$

From (4.151), we have

$$\mathrm{fid}(strexp) = \mathrm{fid}(strexp') \cup \mathrm{fid}(sigexp) \tag{4.169}$$

Moreover, from Proposition 4.3.5 and from assumptions and from (4.169), we have

$$B' \sqsupseteq (B \downarrow \mathrm{fid}(strexp')) \tag{4.170}$$
$$B' \sqsupseteq (B \downarrow \mathrm{fid}(sigexp) \tag{4.171}$$

Also, it follows from assumptions and the definition of agreement that

$$B' \approx_{\mathrm{fid}(\mathit{strexp}')} B \tag{4.172}$$

We can now apply induction to (4.163), (4.170), and (4.172) to get

$$B' \vdash \mathit{strexp}' \Rightarrow (T)E \tag{4.173}$$

Further, from Proposition 4.6.2 and from (4.164) and (4.171), we have

$$B' \vdash \mathit{sigexp} \Rightarrow \Sigma' \tag{4.174}$$

It now follows from rule 2.25 and from (4.173), (4.174), (4.165), (4.168), and (4.167) that we have $B' \vdash \mathit{strexp} \Rightarrow \Sigma$, as required.

CASE $\mathit{strexp} = \mathit{funid}$ (strexp') From assumptions and from rule 2.27, we have

$$B \vdash \mathit{strexp}' \Rightarrow (T)E \tag{4.175}$$
$$B(\mathit{funid}) \geq (E'', (T')E') \tag{4.176}$$
$$E \succ E'' \tag{4.177}$$
$$(T \cup T') \cap \mathrm{tynames}\ B = \emptyset \tag{4.178}$$
$$\Sigma = (T \cup T')E' \tag{4.179}$$

By appropriate renaming of bound names of Σ, we can assume

$$(T \cup T') \cap \mathrm{tynames}\ B' = \emptyset \tag{4.180}$$

From (4.152), we have

$$\mathrm{fid}(\mathit{strexp}) = \{\mathit{funid}\} \cup \mathrm{fid}(\mathit{strexp}') \tag{4.181}$$

Now, from Proposition 4.3.5 and from assumptions and from (4.181), we have

$$B' \sqsupseteq (B \downarrow \mathrm{fid}(\mathit{strexp}')) \tag{4.182}$$

Moreover, from assumptions and from the definition of agreement and from (4.181), we have

$$B' \approx_{\mathrm{fid}(\mathit{strexp}')} B \tag{4.183}$$

We can now apply induction to (4.175), (4.182), and (4.183) to get

$$B' \vdash strexp' \Rightarrow (T)E \tag{4.184}$$

Also, from Proposition 4.3.7, from assumptions, and from (4.181), we have $B'(funid) = B(funid)$, hence from (4.176), we have

$$B'(funid) \geq (E'', (T')E') \tag{4.185}$$

It follows from rule 2.27 and from (4.184), (4.185), (4.177), (4.180), and (4.179) that $B' \vdash strexp \Rightarrow \Sigma$, as required.

CASE $strdec = dec$ From assumptions and from rule 2.28, we have

$$E \vdash dec \Rightarrow \Sigma \tag{4.186}$$
$$B = (F, E) \tag{4.187}$$

Let $B' = (F', E')$. From assumptions and from (4.154) and from the definitions of restriction and strong enrichment, we have

$$E' \sqsupseteq (E \downarrow \mathrm{fid}(dec)) \tag{4.188}$$

Moreover, from assumptions and the definition of agreement, we have

$$E' \approx_{\mathrm{fid}(dec)} E \tag{4.189}$$

Now, from Proposition 4.5.4 and from (4.186), (4.188), and (4.189), we have

$$E' \vdash dec \Rightarrow \Sigma \tag{4.190}$$

From rule 2.28 and from (4.190), we have $B' \vdash strdec \Rightarrow \Sigma$, as required.

CASE $strdec = strdec_1\ strdec_2$ From assumptions and from rule 2.30, we have

$$B \vdash strdec_1 \Rightarrow (T_1)E_1 \tag{4.191}$$
$$B + E_1 \vdash strdec_2 \Rightarrow (T_2)E_2 \tag{4.192}$$
$$(T_1 \cup T_2) \cap \mathrm{tynames}\ B = \emptyset \tag{4.193}$$
$$T_2 \cap (T_1 \cup \mathrm{tynames}\ E_1) = \emptyset \tag{4.194}$$
$$\Sigma = (T_1 \cup T_2)(E_1 + E_2) \tag{4.195}$$

By renaming of bound names of Σ, we can assume

$$(T_1 \cup T_2) \cap \text{tynames } B' = \emptyset \tag{4.196}$$

From Proposition 4.7.1 and from (4.191), we have $\text{decl}(\mathit{strdec}_1) = \text{Dom } E_1$, hence from (4.156), we have

$$\text{fid}(\mathit{strdec}) = \text{fid}(\mathit{strdec}_1) \cup (\text{fid}(\mathit{strdec}_2) \setminus\setminus \text{Dom } E_1) \tag{4.197}$$

Now, from Proposition 4.3.5 and from assumptions and from (4.197), we have

$$B' \sqsupseteq (B \downarrow \text{fid}(\mathit{strdec}_1)) \tag{4.198}$$

$$B' \sqsupseteq (B \downarrow (\text{fid}(\mathit{strdec}_2) \setminus\setminus \text{Dom } E_1)) \tag{4.199}$$

Moreover, from assumptions and from (4.197) and from the definition of agreement, we have

$$B' \approx_{\text{fid}(\mathit{strdec}_1)} B \tag{4.200}$$

We can now apply induction to (4.191), (4.198), and (4.200) to get

$$B' \vdash \mathit{strdec}_1 \Rightarrow (T_1)E_1 \tag{4.201}$$

Now, from Proposition 4.7.1 and from (4.192), we have

$$\text{LongIds}(B + E_1) \supseteq (\text{fid}(\mathit{strdec}_2) \setminus \text{LongStrId}) \tag{4.202}$$

From Proposition 4.3.3 and from (4.199) and (4.202), we have

$$(B' + E_1) \sqsupseteq ((B + E_1) \downarrow \text{fid}(\mathit{strdec}_2)) \tag{4.203}$$

Moreover, from assumptions and from (4.197) and from the definitions of agreement and curtailment, we have

$$(B' + E_1) \approx_{\text{fid}(\mathit{strdec}_2)} (B + E_1) \tag{4.204}$$

We can now apply induction to (4.192), (4.203), and (4.204) to get

$$B' + E_1 \vdash \mathit{strdec}_2 \Rightarrow (T_2)E_2 \tag{4.205}$$

From rule 2.30 and from (4.201), (4.196), (4.205), (4.194), and (4.195), we have $B' \vdash \mathit{strdec} \Rightarrow \Sigma$, as required.

The proofs for the remaining cases are similar. □

Top-level Declarations $\boxed{\text{fid}(topdec) = L}$

$$\begin{aligned}
\text{fid}(strdec) &= \text{fid}(strdec) && (4.206)\\
\text{fid}(\texttt{ functor } funid \texttt{ (} strid \texttt{ : } sigexp \texttt{) } \texttt{= } strexp \texttt{)} &= \text{fid}(sigexp) \cup (\text{fid}(strexp) \setminus\setminus \{strid\}) && (4.207)\\
\text{fid}(topdec_1\ topdec_2) &= \text{fid}(topdec_1) \cup (\text{fid}(topdec_2) \setminus\setminus \text{decl}(topdec_1)) && (4.208)\\
\text{fid}(\varepsilon) &= \emptyset && (4.209)
\end{aligned}$$

$\boxed{\text{decl}(topdec) = I}$

$$\begin{aligned}
\text{decl}(strdec) &= \text{decl}(strdec) && (4.210)\\
\text{decl}(\texttt{ functor } funid \texttt{ (} strid \texttt{ : } sigexp \texttt{) } \texttt{= } strexp) &= \{funid\} && (4.211)\\
\text{decl}(topdec_1\ topdec_2) &= \text{decl}(topdec_1) \cup \text{decl}(topdec_2) && (4.212)\\
\text{decl}(\varepsilon) &= \emptyset && (4.213)
\end{aligned}$$

Proposition 4.7.3 *If* $B \vdash topdec \Rightarrow (T)B'$ *then* $\text{LongIds}(B) \supseteq (\text{fid}(topdec) \setminus \text{LongStrId})$ *and* $\text{decl}(topdec) = \text{Dom}\ B'$.

PROOF By induction over the structure of *topdec*. □

Elaboration of top-level declarations depends on assumptions for those long identifiers that occur free in the phrase.

Proposition 4.7.4 (Top-level dependence) *If* $B \vdash topdec \Rightarrow (T)B''$ *and* $B' \sqsupseteq (B \downarrow \text{fid}(topdec))$ *and* $B' \approx_{\text{fid}(topdec)} B$ *then* $B' \vdash topdec \Rightarrow (T)B''$.

PROOF The proof is by induction over the structure of *topdec* and proceeds by case analysis.

$\boxed{\text{CASE } topdec = strdec}$ From assumptions and from rule 2.32, we have $B \vdash topdec \Rightarrow (T)(\{\}, E)$ and $B \vdash strdec \Rightarrow (T)E$. From (4.206), we have $\text{fid}(topdec) = \text{fid}(strdec)$, hence, from assumptions and from Proposition 4.7.2, we have $B' \vdash strdec \Rightarrow (T)E$, hence, from rule 2.32, we have $B' \vdash topdec \Rightarrow (T)(\{\}, E)$, as required.

CASE $topdec = \texttt{functor}\ funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ strexp$ From assumptions and from rule 2.33, we have

$$B \vdash sigexp \Rightarrow (T)E \tag{4.214}$$
$$T \cap \text{tynames}\ B = \emptyset \tag{4.215}$$
$$B + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma \tag{4.216}$$
$$F = \{funid \mapsto (T)(E, \Sigma) \tag{4.217}$$
$$B \vdash topdec \Rightarrow (\emptyset)(F, \{\}) \tag{4.218}$$

By appropriate renaming of bound names of $(T)E$, we can assume

$$T \cap \text{tynames}\ B' = \emptyset \tag{4.219}$$

From (4.207), we have

$$\text{fid}(topdec) = \text{fid}(sigexp) \cup (\text{fid}(strexp) \setminus\setminus \{strid\}) \tag{4.220}$$

From Proposition 4.3.5, from assumptions, and from (4.220), we have

$$B' \sqsupseteq (B \downarrow \text{fid}(sigexp)) \tag{4.221}$$
$$B' \sqsupseteq (B \downarrow (\text{fid}(strexp) \setminus\setminus \{strid\})) \tag{4.222}$$

Now, from Proposition 4.6.2, from (4.214), and from (4.221), we have

$$B' \vdash sigexp \Rightarrow (T)E \tag{4.223}$$

From Proposition 4.7.3 and from (4.216), we have

$$\text{LongIds}(B + \{strid \mapsto E\}) \supseteq (\text{fid}(strexp) \setminus \text{LongStrId}) \tag{4.224}$$

Moreover, from Proposition 4.3.3, from (4.222), and from (4.224), we have

$$(B' + \{strid \mapsto E\}) \sqsupseteq ((B + \{strid \mapsto E\}) \downarrow \text{fid}(strexp)) \tag{4.225}$$

From the definitions of agreement and curtailment, from assumptions, and from (4.220), we have

$$(B' + \{strid \mapsto E\}) \approx_{\text{fid}(strexp)} (B + \{strid \mapsto E\}) \tag{4.226}$$

Now, from Proposition 4.7.2 and from (4.216), (4.225), and (4.226), we have

$$B' + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma \tag{4.227}$$

From rule 2.33 and from (4.223), (4.219), (4.227), and (4.217), we have $B' \vdash topdec \Rightarrow (\emptyset)(F, \{\})$, as required.

CASE $topdec = topdec_1\ topdec_2$ From assumptions and from rule 2.34, we have

$$B \vdash topdec_1 \Rightarrow (T_1)E_1 \tag{4.228}$$
$$(T_1 \cup T_2) \cap \text{tynames } B = \emptyset \tag{4.229}$$
$$B + B_1 \vdash topdec_2 \Rightarrow (T_2)E_2 \tag{4.230}$$
$$T_2 \cap (T_1 \cup \text{tynames } B_1) = \emptyset \tag{4.231}$$
$$B \vdash topdec \Rightarrow (T_1 \cup T_2)(B_1 + B_2) \tag{4.232}$$

By appropriate renaming of bound names of $(T_1 \cup T_2)(B_1 + B_2)$, we can assume

$$(T_1 \cup T_2) \cap \text{tynames } B' = \emptyset \tag{4.233}$$

From (4.208), from Proposition 4.7.3, and from (4.228), we have

$$\text{fid}(topdec) = \text{fid}(topdec_1) \cup (\text{fid}(topdec_2) \setminus\setminus \text{Dom } B_1) \tag{4.234}$$

Now, from Proposition 4.3.5, from assumptions, and from (4.234), we have

$$B' \sqsupseteq (B \downarrow \text{fid}(topdec_1)) \tag{4.235}$$
$$B' \sqsupseteq (B \downarrow (\text{fid}(topdec_2) \setminus\setminus \text{Dom } B_1)) \tag{4.236}$$

Moreover, from assumptions, from (4.234), and from the definition of agreement, we have

$$B' \approx_{\text{fid}(topdec_1)} B \tag{4.237}$$

We can now apply induction to (4.228), (4.235), and (4.237), to get

$$B' \vdash topdec_1 \Rightarrow (T_1)B_1 \tag{4.238}$$

From Proposition 4.7.3 and from (4.230), we have $\text{LongIds}(B + B_1) \supseteq (\text{fid}(topdec_2) \setminus \text{LongStrId})$, hence, from Proposition 4.3.3 and from (4.236), we have

$$(B' + B_1) \sqsupseteq ((B + B_1) \downarrow \text{fid}(topdec_2)) \tag{4.239}$$

Moreover, from assumptions, from (4.234), and from the definitions of agreement and curtailment, we have

$$(B' + B_1) \approx_{\mathrm{fid}(topdec_2)} (B + B_1) \tag{4.240}$$

We can now apply induction to (4.230), (4.239), and (4.240), to get

$$B' + B_1 \vdash topdec_2 \Rightarrow (T_2)B_2 \tag{4.241}$$

Now, from rule 2.34 and from (4.238), (4.233), (4.241), (4.231), and (4.232), we have $B' \vdash topdec \Rightarrow (T_1 \cup T_2)(B_1 + B_2)$, as required.

CASE $topdec = \varepsilon$ From rule 2.35, we have $B' \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \{\})$, as required. □

4.8 Refinement of Elaboration Dependence

There seem to be a number of ways in which to improve elaboration dependence. Consider the program

```
local structure S = A
in val a = S.j
end
```

This program depends only on assumptions for the long value identifier `A.j`, but elaboration dependence suggests that the program depends on assumptions for `A`. This means that a separate compilation system based on elaboration dependence, as we have defined it here, will force more files to be reelaborated (and thus recompiled) than necessary.

Another problem is the special care that we must take to ensure that the identifier status for free value identifiers in patterns do not change; if the set of top-level value constructors is changed for some program unit then all program units that depends on it must be reelaborated (and thus recompiled). From a language design point of view, it has been suggested elsewhere [Mac92, App93] to make it possible to distinguish syntactically between value variables and value constructors (e.g., by requiring constructor variables to start with an uppercase letter); such an approach is taken by Haskell and Prolog. If it is possible to distinguish syntactically between value variables and value constructors then the notion of identifier status can be eliminated from the static semantics; elaboration dependence can thus be simplified, as well.

Chapter 5

From Opaque to Transparent Modules

In ModML, there are two mechanisms by which the implementation of a type constructor can be hidden by the programmer – by functor abstraction and by using opaque signature constraints. In Chapter 8, we demonstrate that it is possible to translate ModML programs that do not contain opaque signature constraints into an explicitly typed intermediate language called IntML, by flattening structures and specialising ModML functors for each application. The IntML language has no notion of Modules and neither does it support abstract types.

In this chapter, we present a translation for eliminating opaque signature constraints in ModML program phrases by translating opaque signature constraints into transparent signature constraints. This means that a compiler can compile ModML, including opaque signature constraints, into the explicitly typed intermediate language IntML. Although the translation is straightforward, it is not trivial to demonstrate that the translation preserves elaboration. The main problem is that generativity is affected by the translation. Consider the functor declaration

```
functor f() = struct type a = int end :> sig type a end
```

and let $B_0 = \{\texttt{int} \mapsto (t_{\text{int}}, \{\})\}$, where t_{int} is a type name with arity 0. Then the functor declaration for `f` elaborates, in the basis B_0, to a program signature that contains the functor signature

$$(\emptyset)(\{\}, (\{t\})\{\texttt{a} \mapsto (t, \{\})\})$$

for `f`, where t is a type name with arity 0. The functor declaration translates into the functor declaration

```
functor f() = struct type a = int end : sig type a end
```

which, in the basis B_0, elaborates to a program signature that contains the following functor signature for `f`:

$$(\emptyset)(\{\}, (\emptyset)\{\mathtt{a} \mapsto (t_{\mathrm{int}}, \{\})\})$$

In this case, generativity (or type abstraction) has decreased (i.e., after the translation, the type constructor `a` is known to stand for the type denoted by the type name t_{int}.)

As another example, consider the functor declaration

```
functor g() = struct datatype a = A
                     type b = a
              end :> sig type a type b end
```

which elaborates, in the empty basis, to a program signature containing the functor signature

$$(\emptyset)(\{\}, (\{t, t'\})\{\mathtt{a} \mapsto (t, \{\}), \mathtt{b} \mapsto (t', \{\})\})$$

for `g`, where t and t' are type names with arity 0. The functor declaration translates into the functor declaration

```
functor g() = struct datatype a = A
                     type b = a
              end : sig type a type b end
```

which elaborates, in the empty basis, to the program signature containing the following functor signature for `g`:

$$(\emptyset)(\{\}, (\{t\})\{\mathtt{a} \mapsto (t, \{\}), \mathtt{b} \mapsto (t, \{\})\})$$

where t is a type name with arity 0. Again, the translation decreases generativity (or type abstraction).

As a final example illustrating a decrease in generativity, consider the functor declaration

```
functor h(s : sig type a end) =
   struct type b = s.a end :> sig type b end
```

This functor declaration elaborates, in the empty basis, to a program signature containing the functor signature

$$(\{t\})(\{\mathtt{a} \mapsto (t, \{\})\}, (\{t'\})\{\mathtt{b} \mapsto (t', \{\})\})$$

for `h`, where t and t' are type names with arity 0. The functor declaration translates into the functor declaration

```
functor h(s : sig type a end) =
   struct type b = s.a end : sig type b end
```

which elaborates, in the empty basis, to a program signature containing the functor signature

$$(\{t\})(\{\mathtt{a} \mapsto (t, \{\})\}, (\emptyset)\{\mathtt{b} \mapsto (t, \{\})\})$$

where t is a type name with arity 0. Also in this case, the translation decreases generativity.

It is not always the case, however, that the translation decreases generativity (or type abstraction). In fact, the translation may increase generativity. Consider the functor declaration

```
functor i() = struct datatype a = A
                         and b = B
                    type c = a -> b
              end :> sig type c end
```

which elaborates, in the empty basis, to a program signature containing the functor signature

$$(\emptyset)(\{\}, (\{t\})\{\mathtt{c} \mapsto (t, \{\})\})$$

where t is a type name with arity 0. The functor declaration translates into the functor declaration

```
functor i() = struct datatype a = A
                         and b = B
                    type c = a -> b
              end : sig type c end
```

which elaborates, in the empty basis, to a program signature containing the functor signature[1]

$$(\emptyset)(\{\}, (\{t, t'\})\{\mathtt{c} \mapsto (\Lambda().t \to t', \{\})\})$$

where t and t' are type names with arity 0.

The remainder of this chapter is organised as follows. In Section 5.1, we present the translation called opacity elimination for translating opaque signature constraints into transparent ones. Then, in Section 5.2 – inspired by the previous discussion – we present a relation called abstraction. In Section 5.3, we demonstrate that opacity elimination preserves elaboration, under assumptions related by abstraction. In Section 5.4, we discuss why a well-formedness requirement in the elaboration rule for `where type` signature expressions is problematic for opacity elimination.

5.1 Opacity Elimination

The translation translates module phrases, which may contain opaque signature constraints, into module phrases that do not contain opaque signature constraints. We write the translation on the form $oe(phrase) = phrase'$, where *phrase* is the module phrase before the translation and $phrase'$ is the resulting module phrase.

Structure-level Expressions $\boxed{oe(strexp) = strexp'}$

$$\begin{aligned}
oe(\texttt{struct}\ strdec\ \texttt{end}) &= \texttt{struct}\ oe(strdec)\ \texttt{end} && (5.1)\\
oe(longstrid) &= longstrid && (5.2)\\
oe(strexp\ \texttt{:}\ sigexp) &= oe(strexp)\ \texttt{:}\ sigexp && (5.3)\\
oe(strexp\ \texttt{:>}\ sigexp) &= oe(strexp)\ \texttt{:}\ sigexp && (5.4)\\
oe(funid\ \texttt{(}\ strexp\ \texttt{)}) &= funid\ \texttt{(}\ oe(strexp)\ \texttt{)} && (5.5)
\end{aligned}$$

Structure-level Declarations $\boxed{oe(strdec) = strdec'}$

$$oe(dec) = dec \qquad (5.6)$$

[1]The result signature of a functor signature need not be type explicit.

$$oe(\texttt{structure}\ strid\ \texttt{=}\ strexp) = \texttt{structure}\ strid\ \texttt{=}\ oe(strexp) \quad (5.7)$$
$$oe(strdec_1\ strdec_2) = oe(strdec_1)\ oe(strdec_2) \quad (5.8)$$
$$oe(\varepsilon) = \varepsilon \quad (5.9)$$

Top-level Declarations $\boxed{oe(topdec) = topdec'}$

$$oe(strdec) = oe(strdec) \quad (5.10)$$
$$oe(\texttt{functor}\ funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ strexp) = \texttt{functor}\ funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ oe(strexp) \quad (5.11)$$
$$oe(topdec_1\ topdec_2) = oe(topdec_1)\ oe(topdec_2) \quad (5.12)$$
$$oe(\varepsilon) = \varepsilon \quad (5.13)$$

The only interesting equation is (5.4), which translates an opaque signature constraint into a transparent signature constraint.

In the following sections, we demonstrate that elaboration is preserved under opacity elimination in the sense that if a program elaborates under some assumptions B then the translated program elaborates under assumptions that are related to B. This relation, between bases in which the program and the translated program elaborate, was justified in the beginning of this chapter.

5.2 Abstraction

A signature Σ_1 *abstracts* another signature $\Sigma_2 = (T_2)E_2$, written $\Sigma_1 \succeq \Sigma_2$, iff $\Sigma_1 \geq E_2$ and tynames $\Sigma_1 \cap T_2 = \emptyset$. Signature abstraction is essentially the same as signature instantiation, as defined in [MTH90, Section 5.9]. The following proposition states that signature abstraction is closed under realisation:

Proposition 5.2.1 (Signature abstraction is closed under realisation) *If $\Sigma_1 \succeq \Sigma_2$ then $\varphi(\Sigma_1) \succeq \varphi(\Sigma_2)$ for any realisation φ.*

PROOF Let φ be any realisation and let $\Sigma_2 = (T_2)E_2$ such that

$$T_2 \cap (\text{Supp}\ \varphi \cup \text{Yield}\ \varphi) = \emptyset \quad (5.14)$$

From the definition of signature abstraction, we have

$$T_2 \cap \text{tynames } \Sigma_1 = \emptyset \tag{5.15}$$

$$\Sigma_1 \geq E_2 \tag{5.16}$$

Now, from Proposition 3.1.3 and from (5.16), we have

$$\varphi(\Sigma_1) \geq \varphi(E_2) \tag{5.17}$$

It follows from (5.14) and from (5.15) that we have $T_2 \cap \text{tynames}(\varphi(\Sigma_1)) = \emptyset$, hence, from (5.17), from (5.14), and from the definition of signature abstraction, we have $\varphi(\Sigma_1) \succeq \varphi(\Sigma_2)$, as required. □

We extend the notion of abstraction as follows. A functor signature $\Phi_1 = (T_1)(E_1, \Sigma_1)$ *abstracts* another functor signature $\Phi_2 = (T_2)(E_2, \Sigma_2)$, written $\Phi_1 \succeq \Phi_2$, iff $T_1 = T_2$ and $E_1 = E_2$ and $\Sigma_1 \succeq \Sigma_2$. Further, a functor environment F_1 *abstracts* another functor environment F_2, written $F_1 \succeq F_2$, iff Dom F_1 = Dom F_2 and $F_1(\mathit{funid}) \succeq F_2(\mathit{funid})$, for all $\mathit{funid} \in$ Dom F_1. Moreover, a basis $B_1 = (F_1, E_1)$ *abstracts* another basis $B_2 = (F_2, E_2)$, written $B_1 \succeq B_2$, iff $F_1 \succeq F_2$ and $E_1 = E_2$. Finally, a program signature $(T_1)B_1$ *abstracts* another program signature $(T_2)B_2$, written $(T_1)B_1 \succeq (T_2)B_2$, iff there exists a realisation φ such that (1) Supp $\varphi \subseteq T_1$, (2) $\varphi(B_1) \succeq B_2$, and (3) $T_2 \cap \text{tynames}((T_1)B_1) = \emptyset$.

In Section 4.6 we found that elaboration of signature expressions depends on only those identifiers that occur free in the signature expression. As a consequence, elaboration of signature expressions is closed under abstraction. Moreover, in Section 2.12 we found that elaboration of signature expressions is closed under realisation. These two properties, about elaboration of signature expressions, lead to the following proposition:

Proposition 5.2.2 *If* $B \vdash \mathit{sigexp} \Rightarrow \Sigma$ *and* $\varphi(B) \succeq B'$ *then* $B' \vdash \mathit{sigexp} \Rightarrow \varphi(\Sigma)$.

PROOF From Proposition 3.1.10 and from assumptions, we have

$$\varphi(B) \vdash \mathit{sigexp} \Rightarrow \varphi(\Sigma) \tag{5.18}$$

Let $B = (F, E)$ and let $B' = (F', E')$. From assumptions and from the definition of abstraction, we have

$$\varphi(F) \succeq F' \tag{5.19}$$

$$\varphi(E) = E' \tag{5.20}$$

From Proposition 4.6.1 and from (5.18), we have $\mathrm{LongIds}(\varphi(B)) \supseteq \mathrm{fid}(sigexp)$. Because $\mathrm{fid}(sigexp) \cap \mathrm{FunId} = \emptyset$, it follows that

$$\mathrm{LongIds}(\varphi(E)) \supseteq \mathrm{fid}(sigexp) \tag{5.21}$$

Now, from Proposition 4.2.1 and from (5.21), we have that there exists an environment E_0 such that

$$E_0 = (\varphi(E)) \downarrow \mathrm{fid}(sigexp) \tag{5.22}$$

Moreover, from Proposition 4.3.1 and from (5.22), we have $E' \sqsupseteq ((\varphi(E)) \downarrow \mathrm{fid}(sigexp))$. It follows from the definitions of restriction and strong enrichment and because $\mathrm{fid}(sigexp) \cap \mathrm{FunId} = \emptyset$ that we have

$$B' \sqsupseteq ((\varphi(B)) \downarrow \mathrm{fid}(sigexp)) \tag{5.23}$$

From Proposition 4.6.2 and from (5.18) and (5.23), we have $B' \vdash sigexp \Rightarrow \varphi(\Sigma)$, as required. □

5.3 Preservation of Elaboration

We shall now see that elaboration of any module phrase is preserved under opacity elimination. This property is stated in the following proposition:

Proposition 5.3.1 (Preservation of elaboration) *Let phrase be either a structure-level expression, a structure-level declaration, or a top-level declaration. If $B \vdash phrase \Rightarrow A$ and $\varphi(B) \succeq B'$ then there exists a semantic object A' such that $\varphi(A) \succeq A'$ and $B' \vdash oe(phrase) \Rightarrow A'$.*

PROOF The proof is by induction over the structure of *phrase*. The proof proceeds by case analysis; there is one case for each applicable elaboration rule. The proofs for the cases corresponding to rule 2.23, rule 2.29, rule 2.31, rule 2.32, and rule 2.35 follow either immediately or immediately by induction. The proofs for the remaining cases follow.

CASE $strexp = longstrid$ From assumptions and rule 2.24, we have

$$\begin{aligned} B(longstrid) &= E \\ B &\vdash strexp \Rightarrow (\emptyset)E \end{aligned} \tag{5.24}$$

From assumptions, from the definition of abstraction, and from (5.24), we have $B'(longstrid) = \varphi(E)$. Let $\Sigma' = \varphi((\emptyset)E)$. It follows from rule 2.24 and from (5.2) that $B' \vdash oe(strexp) \Rightarrow \Sigma'$. Now, from the definition of abstraction, we have $\varphi((\emptyset)E) \succeq \Sigma'$, as required.

CASE $strexp = strexp' : sigexp$ From assumptions and from rule 2.25, we have

$$B \vdash strexp' \Rightarrow (T)E \tag{5.25}$$
$$B \vdash sigexp \Rightarrow \Sigma \tag{5.26}$$
$$\Sigma \geq E' \prec E \tag{5.27}$$
$$B \vdash strexp \Rightarrow (T)E' \tag{5.28}$$
$$T \cap \text{tynames } B = \emptyset \tag{5.29}$$

Because of side condition (5.29), we can assume that the bound names of $(T)E'$ have been chosen such that

$$T \cap (\text{Supp } \varphi \cup \text{Yield } \varphi \cup \text{tynames } B') = \emptyset \tag{5.30}$$

From assumptions we have $\varphi(B) \succeq B'$, hence from (5.25), we can now apply induction to get there exists a signature $\Sigma_1 = (T_1)E_1$ such that

$$(T)(\varphi(E)) \succeq \Sigma_1 \tag{5.31}$$
$$B' \vdash oe(strexp') \Rightarrow \Sigma_1 \tag{5.32}$$

We are free to choose the bound names of Σ_1 such that

$$T_1 \cap (\text{tynames } B' \cup \text{tynames}(\varphi((T)E'))) = \emptyset \tag{5.33}$$

From (5.31) and from the definitions of abstraction and signature instantiation, we have that there exists a realisation φ' such that

$$(\varphi' \circ \varphi)(E) = E_1 \tag{5.34}$$
$$\text{Supp } \varphi' \subseteq T \tag{5.35}$$
$$T_1 \cap \text{tynames}((T)(\varphi(E))) = \emptyset \tag{5.36}$$

From Proposition 3.1.7 and from (5.27) and (5.34), we have

$$(\varphi' \circ \varphi)(E') \prec E_1 \tag{5.37}$$

Moreover, from Proposition 3.1.3 and from (5.27), we have

$$(\varphi' \circ \varphi)(\Sigma) \geq (\varphi' \circ \varphi)(E') \tag{5.38}$$

Further, from Proposition 5.2.2 ,from assumptions, and from (5.26), we have

$$B' \vdash \mathit{sigexp} \Rightarrow (\varphi' \circ \varphi)(\Sigma) \tag{5.39}$$

Now, let $\Sigma' = (T_1)((\varphi' \circ \varphi)(E'))$. From rule 2.25 and from (5.32), (5.39), (5.38), (5.37), (5.30), and (5.3), we have $B' \vdash \mathit{oe}(\mathit{strexp}) \Rightarrow \Sigma'$. Also, from the definitions of abstraction and signature instantiation and from (5.30), we have $\varphi((T)E) \succeq \Sigma'$, as required.

CASE $\mathit{strexp} = \mathit{strexp}'$:> sigexp From assumptions and from rule 2.26, we have

$$B \vdash \mathit{strexp}' \Rightarrow (T)E \tag{5.40}$$
$$B \vdash \mathit{sigexp} \Rightarrow \Sigma \tag{5.41}$$
$$\Sigma \geq E' \prec E \tag{5.42}$$
$$T \cap \text{tynames } B = \emptyset \tag{5.43}$$
$$B \vdash \mathit{strexp} \Rightarrow \Sigma \tag{5.44}$$

From Proposition 3.1.10 and from (5.30), we have

$$\text{tynames } \Sigma \subseteq \text{tynames } B \tag{5.45}$$

Because of side condition (5.43) and from (5.45), we can assume that T has been chosen such that

$$T \cap (\text{Supp } \varphi \cup \text{Yield } \varphi \cup \text{tynames } B') = \emptyset \tag{5.46}$$

From assumptions we have $\varphi(B) \succeq B'$, hence from (5.40), we can now apply induction to get there exists a signature $\Sigma_1 = (T_1)E_1$ such that

$$(T)(\varphi(E)) \succeq \Sigma_1 \tag{5.47}$$
$$B' \vdash \mathit{oe}(\mathit{strexp}') \Rightarrow \Sigma_1 \tag{5.48}$$

From (5.47) and from the definitions of abstraction and signature instantiation, we have that there exists a realisation φ' such that

$$(\varphi' \circ \varphi)(E) = E_1 \tag{5.49}$$
$$\text{Supp } \varphi' \subseteq T \tag{5.50}$$
$$T_1 \cap \text{tynames}((T)(\varphi(E))) = \emptyset \tag{5.51}$$

By appropriate renaming of bound names of Σ_1, we can further assume that

$$T_1 \cap \text{tynames } \varphi(\Sigma) = \emptyset \tag{5.52}$$

From Proposition 3.1.7 and from (5.42) and (5.49), we have

$$(\varphi' \circ \varphi)(E') \prec E_1 \tag{5.53}$$

Moreover, from Proposition 3.1.3 and from (5.42), we have

$$(\varphi' \circ \varphi)(\Sigma) \geq (\varphi' \circ \varphi)(E') \tag{5.54}$$

Further, from Proposition 5.2.2, from assumptions, and from (5.41), we have

$$B' \vdash \mathit{sigexp} \Rightarrow (\varphi' \circ \varphi)(\Sigma) \tag{5.55}$$

Now, let $\Sigma' = (T_1)((\varphi' \circ \varphi)(E''))$. From rule 2.25 and from (5.48), (5.55), (5.54), (5.53), (5.46), and (5.3), we have

$$B' \vdash \mathit{oe}(\mathit{strexp}) \Rightarrow \Sigma'$$

We must now demonstrate that $\varphi(\Sigma) \succeq \Sigma'$. Let $\Sigma = (T')E'$. By appropriate renaming of bound names of Σ, we can assume

$$T' \cap T = \emptyset \tag{5.56}$$

From (5.54) and from the definitions of abstraction and signature instantiation, we have that there exists a realisation φ'' such that

$$\text{Supp } \varphi'' \subseteq T' \tag{5.57}$$

$$(\varphi'' \circ \varphi' \circ \varphi)(E') = (\varphi' \circ \varphi)(E'') \tag{5.58}$$

Moreover, from (5.45), from (5.56), and from (5.43), we have tynames $E' \cap T = \emptyset$, hence from (5.46), we have

$$\text{tynames}(\varphi(E')) \cap T = \emptyset \tag{5.59}$$

Now, from (5.50) and from (5.59), we have

$$(\varphi'' \circ \varphi)(E') = (\varphi' \circ \varphi)(E'') \tag{5.60}$$

It follows from the definition of signature instantiation, from (5.60), and from (5.50) that we have

$$\varphi(\Sigma) \geq (\varphi' \circ \varphi)(E'') \tag{5.61}$$

Finally, from the definition of abstraction, from (5.61), and from (5.52), we have $\varphi(\Sigma) \succeq \Sigma'$, as required.

CASE *strexp* = *funid* (*strexp'*) From assumptions and from rule 2.27, we have

$$B \vdash \mathit{strexp}' \Rightarrow (T)E \tag{5.62}$$
$$\Phi \geq (E'', (T')E') \tag{5.63}$$
$$B(\mathit{funid}) = \Phi \tag{5.64}$$
$$E \succ E'' \tag{5.65}$$
$$(T' \cup T) \cap \text{tynames } B = \emptyset \tag{5.66}$$
$$\Sigma = (T \cup T')E' \tag{5.67}$$
$$B \vdash \mathit{strexp} \Rightarrow \Sigma \tag{5.68}$$

Because of side condition (5.66), we can assume that the bound names of Σ have been chosen such that

$$(T \cup T') \cap (\text{tynames } B' \cup \text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{5.69}$$

From assumptions and from (5.62), we can apply induction to get there exists a signature $\Sigma_1 = (T_1)E_1$ such that

$$(T)(\varphi(E)) \succeq \Sigma_1 \tag{5.70}$$
$$B' \vdash \mathit{oe}(\mathit{strexp}') \Rightarrow \Sigma_1 \tag{5.71}$$

By appropriate renaming of bound names of Σ_1, we can assume

$$T_1 \cap (\text{tynames } B' \cup \text{tynames}(\varphi(\Sigma)) = \emptyset \tag{5.72}$$

From (5.70) and from the definitions of signature instantiation and abstraction, we have that there exists a realisation φ_0 such that

$$\text{Supp } \varphi_0 \subseteq T \tag{5.73}$$
$$(\varphi_0 \circ \varphi)(E) = E_1 \tag{5.74}$$
$$T_1 \cap \text{tynames}((T)(\varphi(E))) = \emptyset \tag{5.75}$$

Moreover, from assumptions and from the definition of abstraction and from (5.64),we have that there exists a functor signature Φ' such that

$$\varphi(\Phi) \succeq \Phi' \tag{5.76}$$
$$B'(\mathit{funid}) = \Phi' \tag{5.77}$$

Now, let $\Phi = (T_0)(E_0, \Sigma'_0)$ and let $\Phi' = (T_2)(E_2, \Sigma'_2)$. From (5.76) and from the definition of abstraction, we have

$$T_0 = T_2 \tag{5.78}$$
$$\varphi(E_0) = E_2 \tag{5.79}$$
$$\varphi(\Sigma'_0) \succeq \Sigma'_2 \tag{5.80}$$

From Proposition 3.1.4 and from (5.63) and (5.69), we have

$$\varphi(\Phi) \geq (\varphi(E''), (T')(\varphi(E'))) \tag{5.81}$$

From (5.81) and from the definition of functor signature instantiation, we have that there exists a realisation φ_1 such that

$$\text{Supp } \varphi_1 \subseteq T_0 \tag{5.82}$$
$$(\varphi_1 \circ \varphi)(E_0) = \varphi(E'') \tag{5.83}$$
$$(\varphi_1 \circ \varphi)(\Sigma'_0) = (T')(\varphi(E')) \tag{5.84}$$

From (5.78), from (5.79), we have $\Phi' = (T_0)(\varphi(E_0), \Sigma'_2)$, hence from (5.82), from (5.83), and from the definition of functor signature instantiation, we have

$$\Phi' \geq (\varphi(E''), \varphi_1(\Sigma'_2)) \tag{5.85}$$

Moreover, from Proposition 3.1.4 and from (5.85), (5.69), (5.73), and (5.77), we have

$$\Phi' \geq ((\varphi_0 \circ \varphi)(E''), (\varphi_0 \circ \varphi_1)(\Sigma'_2)) \tag{5.86}$$

From (5.65) and from Proposition 3.1.7, we have $(\varphi_0 \circ \varphi)(E) \succ (\varphi_0 \circ \varphi)(E'')$, hence from (5.74), we have

$$E_1 \succ (\varphi_0 \circ \varphi)(E'') \tag{5.87}$$

Now, let $\Sigma_2' = (T_2')E_2'$. By appropriate renaming of bound names of Σ_2', we can assume

$$T_2' \cap (\mathrm{Supp}(\varphi_0 \circ \varphi_1) \cup \mathrm{Yield}(\varphi_0 \circ \varphi_1)) = \emptyset \tag{5.88}$$

$$T_2' \cap (\mathrm{tynames}\ B' \cup \mathrm{tynames}(\varphi(\Sigma))) = \emptyset \tag{5.89}$$

From (5.86), (5.88), and (5.77), we have

$$B'(\mathit{funid}) \geq ((\varphi_0 \circ \varphi)(E''), (T_2')((\varphi_0 \circ \varphi_1)(E_2'))) \tag{5.90}$$

Let $\Sigma' = (T_1 \cup T_2')((\varphi_0 \circ \varphi_1)(E_2'))$. From rule 2.27, from (5.71), from (5.90), from (5.87), and from (5.5), we have

$$B' \vdash \mathit{oe}(\mathit{strexp}) \Rightarrow \Sigma' \tag{5.91}$$

We must now show $\varphi(\Sigma) \succeq \Sigma'$. From Proposition 5.2.1, from (5.80), and from (5.84), we have

$$(T')(\varphi(E')) \succeq \varphi_1(\Sigma_2') \tag{5.92}$$

Now, from (5.88), from (5.92), and from the definitions of signature instantiation and abstraction, we have that there exists a realisation φ_2 such that

$$T_2' \cap \mathrm{tynames}((T')(\varphi(E'))) = \emptyset \tag{5.93}$$

$$\mathrm{Supp}\ \varphi_2 \subseteq T' \tag{5.94}$$

$$(\varphi_2 \circ \varphi)(E') = \varphi_1(E_2') \tag{5.95}$$

From (5.95), we have $(\varphi_0 \circ \varphi_2 \circ \varphi)(E') = (\varphi_0 \circ \varphi_1)(E_2')$. Now, from (5.69), from (5.67), from the definition of signature instantiation, and because $\mathrm{Supp}(\varphi_0 \circ \varphi_2) \subseteq (T' \cup T)$ follows from (5.94) and (5.73), we have

$$\varphi(\Sigma) \geq (\varphi_0 \circ \varphi_1)(E_2') \tag{5.96}$$

Moreover, from (5.89) and from (5.72), we have

$$(T_1 \cup T_2') \cap \mathrm{tynames}(\varphi(\Sigma)) = \emptyset \tag{5.97}$$

From (5.96) and from (5.97) and from the definition of signature abstraction, we have $\varphi(\Sigma) \succeq \Sigma'$, as required.

CASE $strdec = dec$ From assumptions and from rule 2.28, we have

$$E \text{ of } B \vdash dec \Rightarrow \Sigma \tag{5.98}$$

From assumptions and from the definition of abstraction, we have $\varphi(E \text{ of } B) = E \text{ of } B'$, hence, from Proposition 3.1.9 and from (5.98), we have $(E \text{ of } B') \vdash dec \Rightarrow \varphi(\Sigma)$. It follows from rule 2.28 and from (5.6) that $B' \vdash oe(strdec) \Rightarrow \varphi(\Sigma)$. From the definition of abstraction, it follows that $\varphi(\Sigma) \succeq \varphi(\Sigma)$, as required.

CASE $strdec = strdec_1\ strdec_2$ The proof for this case is similar to the proof for the case where $topdec = topdec_1\ topdec_2$.

CASE $topdec =$ `functor` $funid$ `(` $strid$ `:` $sigexp$ `)` `=` $strexp$ From assumptions and from rule 2.33, we have

$$B \vdash sigexp \Rightarrow (T)E \tag{5.99}$$
$$T \cap \text{tynames } B = \emptyset \tag{5.100}$$
$$B + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma \tag{5.101}$$
$$F = \{funid \mapsto (T)(E, \Sigma)\} \tag{5.102}$$

By appropriate renaming of bound names, we can assume

$$T \cap (\text{tynames } B' \cup \text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{5.103}$$

From Proposition 5.2.2, from assumptions, from (5.99), and from (5.103), we have

$$B' \vdash sigexp \Rightarrow (T)(\varphi(E)) \tag{5.104}$$

It follows from assumptions and from the definition of abstraction that we have

$$\varphi(B + \{strid \mapsto E\}) \succeq (B' + \{strid \mapsto \varphi(E)\}) \tag{5.105}$$

We can now apply induction to (5.101) and (5.105) to get there exists a signature Σ' such that

$$(B' + \{strid \mapsto \varphi(E)\}) \vdash oe(strexp) \Rightarrow \Sigma' \tag{5.106}$$
$$\varphi(\Sigma) \succeq \Sigma' \tag{5.107}$$

From (5.102), from (5.107), and from the definition of abstraction, we have

$$\varphi(F) \succeq F' \tag{5.108}$$
$$F' = \{funid \mapsto (T)(\varphi(E), \Sigma')\} \tag{5.109}$$

Now, from rule 2.33 and from (5.104), (5.103), (5.106), and (5.109), we have $B' \vdash oe(topdec) \Rightarrow (\emptyset)(F', \{\})$. Moreover, from (5.108) and from the definition of abstraction, we have $\varphi((\emptyset)(F, \{\})) \succeq (\emptyset)(F', \{\})$, as required.

CASE $topdec = topdec_1\ topdec_2$ From assumptions and from rule 2.34, we have

$$B \vdash topdec_1 \Rightarrow (T_1)B_1 \tag{5.110}$$
$$(T_1 \cup T_2) \cap \text{tynames } B = \emptyset \tag{5.111}$$
$$B + B_1 \vdash topdec_2 \Rightarrow (T_2)B_2 \tag{5.112}$$
$$T_2 \cap (T_1 \cup \text{tynames } B_1) = \emptyset \tag{5.113}$$

By appropriate renaming of bound names of $(T_1)B_1$, we can assume

$$T_1 \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset \tag{5.114}$$

From assumptions and from (5.110), we can apply induction to get there exists a program signature $(T_1')B_1'$ such that

$$B' \vdash oe(topdec_1) \Rightarrow (T_1')B_1' \tag{5.115}$$
$$\varphi((T_1)B_1) \succeq (T_1')B_1' \tag{5.116}$$

By appropriate renaming of bound names of $(T_1')B_1'$, we can assume

$$T_1' \cap \text{tynames } B' = \emptyset \tag{5.117}$$
$$T_1' \cap \text{tynames}(\varphi((T_1 \cup T_2)(B_1 + B_2))) = \emptyset \tag{5.118}$$

From the definition of abstraction, from (5.116), and from (5.114), we have that there exists a realisation φ_1 such that

$$\text{Supp } \varphi_1 \subseteq T_1 \tag{5.119}$$
$$(\varphi_1 \circ \varphi)(B_1) \succeq B_1' \tag{5.120}$$
$$T_1' \cap \text{tynames}((T_1)(\varphi(B_1))) = \emptyset \tag{5.121}$$

By appropriate renaming of bound names of $(T_2)B_2$, we can assume

$$T_2 \cap (\mathrm{Supp}(\varphi_1 \circ \varphi) \cup \mathrm{Yield}(\varphi_1 \circ \varphi)) = \emptyset \tag{5.122}$$

From (5.114), (5.111), and (5.119), we have $(\varphi_1 \circ \varphi)(B) = \varphi(B)$, hence, from assumptions, we have

$$(\varphi_1 \circ \varphi)(B) \succeq B' \tag{5.123}$$

Now, from the definition of abstraction and from (5.120) and (5.123), we have

$$(\varphi_1 \circ \varphi)(B + B_1) \succeq (B' + B_1) \tag{5.124}$$

We can now apply induction to (5.112) and (5.124) to get there exists a program signature $(T_2')B_2'$ such that

$$(\varphi_1 \circ \varphi)((T_2)B_2) \succeq (T_2')B_2' \tag{5.125}$$
$$(B' + B_1') \vdash oe(topdec_2) \Rightarrow (T_2')B_2' \tag{5.126}$$

By renaming of bound names of $(T_2')B_2'$, we can assume

$$T_2' \cap (T_1' \cup \mathrm{tynames}\ B_1') = \emptyset \tag{5.127}$$
$$T_2' \cap \mathrm{tynames}\ B' = \emptyset \tag{5.128}$$
$$T_2' \cap \mathrm{tynames}(\varphi((T_1 \cup T_2)(B_1 + B_2))) = \emptyset \tag{5.129}$$

From rule 2.34 and from (5.115), (5.117), (5.128), (5.126), (5.127), and (5.12), we have

$$B' \vdash oe(topdec) \Rightarrow (T_1' \cup T_2')(B_1' + B_2') \tag{5.130}$$

We must now show that $\varphi((T_1 \cup T_2)(B_1 + B_2)) \succeq (T_1' \cup T_2')(B_1' + B_2')$. From (5.125), from (5.122), and from the definition of abstraction, we have that there exists a realisation φ_2 such that

$$\mathrm{Supp}\ \varphi_2 \subseteq T_2 \tag{5.131}$$
$$(\varphi_2 \circ \varphi_1 \circ \varphi)(B_2) \succeq B_2' \tag{5.132}$$
$$T_2' \cap \mathrm{tynames}((T_2)((\varphi_1 \circ \varphi)(B_2))) = \emptyset \tag{5.133}$$

Now, from (5.113), (5.122), and (5.131), we have $(\varphi_2 \circ \varphi_1 \circ \varphi)(B_1) = (\varphi_1 \circ \varphi)(B_1)$, hence, from (5.120), we have

$$(\varphi_2 \circ \varphi_1 \circ \varphi)(B_1) \succeq B_1' \tag{5.134}$$

From (5.132), from (5.134), and from the definition of abstraction, we have

$$(\varphi_2 \circ \varphi_1 \circ \varphi)(B_1 + B_2) \succeq (B_1' + B_2') \tag{5.135}$$

It follows from (5.119) and (5.131) that we have

$$\mathrm{Supp}(\varphi_2 \circ \varphi_1) \subseteq (T_1 \cup T_2) \tag{5.136}$$

Moreover, from (5.118) and from (5.129), we have

$$(T_1' \cup T_2') \cap (\varphi((T_1 \cup T_2)(B_1 + B_2))) = \emptyset \tag{5.137}$$

Now, from the definition of abstraction and from (5.135), (5.136), and (5.137), we have $\varphi((T_1 \cup T_2)(B_1 + B_2)) \succeq (T_1' \cup T_2')(B_1' + B_2')$, as required. □

5.4 Well-Formedness

In this section, we shall see that if signature expressions are required to elaborate to well-formed signatures (see Section 2.11 on page 19 for the definition of well-formedness) then opacity elimination does not preserve elaboration. The crucial point is the rule for elaborating `where type` signature expressions. In SML'97, this rule contains a side condition saying that the resulting signature be well-formed. Consider the SML'97 program

```
structure S = struct type s = int -> int
              end :> sig type s end

signature SIG = sig datatype u = A end
                where type u = S.s
```

This program elaborates under SML'97 and it also elaborates under ModML, provided, ModML is extended to allow for signature declarations. However, elaboration of the preceding program under SML'97 is not closed under opacity elimination; the program

```
structure S = struct type s = int -> int
              end : sig type s end

signature SIG = sig datatype u = A end
                where type u = S.s
```

does not elaborate under SML'97, because the signature

$$(\emptyset)\{\mathtt{u} \mapsto (\Lambda().t_{\text{int}} \to t_{\text{int}}, \{A \mapsto t_{\text{int}} \to t_{\text{int}}\})\}$$

is not well-formed.

For ModML (again, extended to allow for signature declarations), elaboration is closed under opacity elimination. Technically, the problem with requiring well-formedness of the resulting signature, in the rule for `where type` signature expressions, is that the requirement destroys the property that elaboration of signature expressions and specifications is closed under realisation.

As discussed in Section 2.12, no real structure (i.e., a structure existing outside of a functor body) can match a non-well-formed signature. Thus, leaving out the well-formedness requirement in rule 2.20 does not contribute to unsoundness of the static semantics. Because a well-formedness requirement in rule 2.20 only restricts what signature expressions elaborate, we can enforce such a check in an implementation – without compromising soundness.

5.5 Complete Elimination of Abstract Types

In this chapter, we have seen that the translation of opaque signature constraints into transparent signature constraints preserves elaboration under related assumptions. Besides this property being reasonable for a language design point of view, the property makes it possible to eliminate type abstraction in the intermediate language of a compiler, by further specialisation of functors, as we shall demonstrate in Chapter 8.

An interesting aspect of the proof, that opacity elimination preserves elaboration, is that the proof is constructive and thus outlines how type information is updated if opacity elimination is performed on an explicitly typed intermediate representation of the program.

Part II

Compiling Program Components

Chapter 6

Cut-Off Incremental Recompilation

A mechanism for separate compilation is essential for a programming environment. Several different schemes for providing separate compilation were discussed in the introduction. In this chapter, we present a framework for separate compilation that we call cut-off incremental recompilation.

A compiler translates program units of some source language to program units in some target language.[1] This mapping from source program units to target program units is often best described—and implemented—as the composition of a series of translation steps where each translation step maps an intermediate representation of the source program unit into another intermediate representation of the source program unit.

Each source program unit may contain free identifiers that refer to declarations in other program units. Thus, compilation is defined with respect to a so-called basis that provides assumptions for free identifiers of a source program unit. In most frameworks for separate compilation, assumptions for free identifiers of a program unit stem from interfaces for those program units on which the program unit depends.[2] Interfaces may either be gener-

[1] In most Standard ML compilers a source program unit is a sequence of top-level declarations contained in a file on the underlying operating system. The Definition of Standard ML [MTHM97] defines a program to be a sequence of top-level declarations and thus does not provide any solution to how programs are organised on the underlying operation system.

[2] As discussed in Section 1.2, smartest recompilation [SA93] builds up assumptions for free identifiers in a program unit from each of the free occurrences of an identifier, thus,

ated by the compiler when program units are compiled or provided by the programmer. Systems that allow the programmer to provide such interfaces are said to support *cut-off* separate compilation because the system thus has a mechanism for compiling a program unit without first compiling those program units on which the program unit depends. However, for a system to be reasonably useful, there is a limit to the amount of type information a programmer can provide in interfaces. Thus, in systems that support cut-off separate compilation, there is a limit to the amount of information for optimisations and analyses that can be propagated across those program unit boundaries for which interfaces are provided by the programmer.

On the other hand, if interfaces are generated by the compiler then it is possible to use different calling conventions, say, for those identifiers that are declared by a program unit. More important, some analyses and optimisations depend critically on information about identifiers of other program units. Region inference is an example of one such analysis; without information about region type schemes for free identifiers, region inference becomes too conservative.

To allow for information about declared identifiers of any intermediate representation of a program unit to propagate to intermediate representations of other program units, we define a basis to be the product of environments for each translation step in the compiler. Then, each environment provides assumptions for free identifiers of the intermediate program unit for the corresponding translation step. Informally, the result of translating an intermediate program unit p_i is a pair of a new intermediate program unit p_{i+1} (possibly, a target program unit) and an environment that maps each declared identifier of p_i to information appropriate for the ith translation step. Thus informally, the result of compiling a source program unit is a pair of a target program unit and a basis.

We do not assume that there is a one-to-one correspondence between identifiers at the level of source program units and identifiers (generated names) at other levels of translation (e.g., machine code labels at the level of target program units.) Instead, translation environments may maintain a mapping from names of some level to names of the next level; this feature allows the framework to be used with Standard ML and its generative treatment of datatypes, for instance.

The framework that we present is simple because we want to study its

---

for smartest recompilation assumptions do not stem from interfaces.

properties; the framework does not build on any tool for finding dependencies between source program units [Blu97, Chapter 4], although, in principle it could. Instead, we assume that the programmer provides a program as a sequence of source program units; informally, the meaning of the program is then the meaning of the program that would arise by concatenating the sequence of program units. To avoid recompiling a program unit when it has not been modified and when the assumptions under which the program unit was last compiled have not changed, the framework has a notion of repository for holding compilation results of compiled program units.

The framework supports cut-off incremental recompilation because compilation of a sequence of program units that form a program must be performed incrementally and because a program unit need not necessarily be recompiled if program units on which it depends have been recompiled. The framework builds on properties of each translation step in a compiler. We describe the framework by assuming a translation function for each translation step in a compiler and by assuming a set of properties of each translation step. Based on these properties, we demonstrate soundness and completeness of the separate compilation framework. Soundness expresses that if a project compiles to some result in some repository that satisfy certain well-formedness conditions then the same project compiles to the same result in an empty repository. Similarly, completeness expresses that if a project compiles to some result in an empty repository then the same project compiles to the same result in any well-formed repository.

In the following sections, we introduce the notion of translation steps and state the properties that must hold for each translation step so as to demonstrate correctness of the separate compilation framework. We then proceed to describe how translation steps are composed to form the notion of compilation. In Section 6.5, we present the separate compilation framework. In Section 6.6, we demonstrate soundness and completeness of the separate compilation framework.

6.1 Translation Environments

We assume a denumerably infinite set Name of names, ranged over by n. For simplicity, we assume that source program identifiers are members of Name. The set of all subsets of Name is written NameSet and we use N to range over NameSet. When A is any object, we write names A to mean the set of

names that occur free in A; we assume that what free means is defined for actual objects used in the framework.

A *(translation) environment* E for a translation step in the compiler is a finite map from names to translation objects; we use to to range over translation objects for some translation step, but we shall not define translation objects here, as such objects are specific for actual translation steps. However, we assume a notion of equality on translation objects; when to_1 and to_2 are translation objects, we write $to_1 = to_2$ iff to_1 equals to_2. The set of names that occur free in an environment E, written names E, is the set Dom $E \cup$ names(Ran E).

We now define a relation on environments called strong enrichment:[3]

Definition 6.1.1 (Strong enrichment) An environment E_1 *strongly enriches* another environment E_2, written $E_1 \sqsupseteq E_2$, iff Dom $E_1 \supseteq$ Dom E_2 and $E_1(n) = E_2(n)$ for all $n \in$ Dom E_2. □

Strong enrichment is reflexive, transitive, and antisymmetric, thus, for a given translation step in the compiler, strong enrichment defines a partial order on environments. Recall that the restriction of a finite map E to a set $N \subseteq$ Dom E, written $E \downarrow N$, is the finite map with domain N and values $(E \downarrow N)(x) = E(x)$ for all $x \in N$.

6.2 Translation Steps

We use p to range over program units (of some translation step) of the compiler. When p is a program unit, we write uses p to mean the set of names that appear as uses in p and we write decls p to mean the set of names that are declared by p; declared names of a program unit do not alpha-vary.

In our framework, each translation step is assumed to be given on the form

$$E \vdash p \Rightarrow (N)(E', p')$$

where E and E' are environments, p is a source program unit for the translation step, p' is a target program unit for the translation step, and N is the set of names that are generated during translation. Sentences of this form are read "p translates to $(N)(E', p')$ in E." The prefix (N) in the object

[3]The term "strong enrichment" stem from the strong enrichment relation in Chapter 4.

$(N)(E', p')$ binds names and we consider objects of this form equal up to renaming of bound names.

The framework assumes a set of properties be met. First, each translation step is assumed to be closed under strong enrichment:

Property 6.2.1 (Translation is closed under strong enrichment) If $E \vdash p \Rightarrow (N)(E', p')$ and $E'' \sqsupseteq (E \downarrow \text{uses } p)$ then $E'' \vdash p \Rightarrow (N)(E', p')$. □

This property guarantees that the translation step depends only on those assumptions for which identifiers occur free in the program unit to be translated; it is this property that allows the result of compiling a program unit to be reused. Informally, the requirement $E'' \sqsupseteq (E \downarrow \text{uses } p)$ expresses that E and E' agree on all uses in p.

Second, used names of a target program unit of a translation step are assumed to stem from propagating uses of the source program unit of the translation step through the translation environment:

Property 6.2.2 (Usage propagation) If $E \vdash p \Rightarrow (N)(E', p')$ then $\text{uses } p' \subseteq \text{names}(\text{Ran}(E \downarrow \text{uses } p))$. □

This property states that if some translation step translates a program unit p to another program unit p' then there is a connection between the uses of p and the uses of p'; we shall return to the importance of this property in Section 6.4.

Finally, each translation step is assumed to be deterministic:

Property 6.2.3 (Translation is deterministic) If $E \vdash p \Rightarrow (N_1)(E_1, p_1)$ and $E \vdash p \Rightarrow (N_2)(E_2, p_2)$ then $(N_1)(E_1, p_1) = (N_2)(E_2, p_2)$. □

It is this property that guarantees completeness of the separate compilation framework.

6.3 Compilation Bases

For the purpose of composing translation steps to form a notion of compilation, we first define a notion of compilation basis. A *(compilation) basis* B is a sequence $E_1.\cdots.E_n$ of one or more translation environments. The translation environment E_i, $1 \leq i \leq n$, in this sequence provides assumptions for

the ith translation step in the compiler. The set of names that occur free in B, written names B, is the set $\text{names}(E_1) \cup \ldots \cup \text{names}(E_n)$.

Strong enrichment is extended to bases. A basis $B = E_1.\cdots.E_n$ *strongly enriches* another basis $B' = E'_1.\cdots.E'_n$, written $B \sqsupseteq B'$, if $E_i \sqsupseteq E'_i$ for all $i \in \{1, \cdots, n\}$.

Strong enrichment on bases is reflexive, transitive, and antisymmetric. These properties follow from the properties of strong enrichment on environments. It follows that strong enrichment on bases defines a partial order on bases.

The *restriction* of a basis B to a name set N, written $B \Downarrow N$, is defined inductively by the following equations:

$$E \Downarrow N = E \downarrow N \tag{6.1}$$

$$(E.B) \Downarrow N = (E \downarrow N).(B \Downarrow (\text{names}(\text{Ran}(E \downarrow N)))) \tag{6.2}$$

Equation 6.2 is best illustrated with a small example. Consider the basis $B = E_1.E_2$ composed by the two environments $E_1 = \{a \mapsto t, b \mapsto s, c \mapsto t\}$ and $E_2 = \{t \mapsto l_1, s \mapsto l_2\}$, where a, b, c, s, t, l_1, and l_2 are names. Then the restriction of B to the name set $\{a\}$ is the basis $\{a \mapsto t\}.\{t \mapsto l_1\}$.

We now demonstrate some properties, which describe the relationship between strong enrichment and restriction.

Proposition 6.3.1 *If $B \sqsupseteq (B' \Downarrow N)$ and $N' \subseteq N$ then $B \sqsupseteq (B' \Downarrow N')$.*

PROOF The proof is by induction on the structure of bases.

CASE $B = E$ The result follows from assumptions and from the definitions of strong enrichment and restriction on environments.

CASE $B = E.B''$ Write B' in the form $E'.B'''$. From assumptions and from the definitions of strong enrichment and restriction, we have

$$E \sqsupseteq (E' \downarrow N) \tag{6.3}$$

$$B'' \sqsupseteq (B''' \Downarrow \text{names}(\text{Ran}(E' \downarrow N))) \tag{6.4}$$

From the definitions of strong enrichment and restriction on environments and from (6.3), we have

$$E \sqsupseteq (E' \downarrow N') \tag{6.5}$$

and because $\mathrm{names}(\mathrm{Ran}(E' \downarrow N')) \subseteq \mathrm{names}(\mathrm{Ran}(E' \downarrow N))$ follows from $N' \subseteq N$, we can apply induction to (6.4) to get

$$B'' \sqsupseteq (B''' \Downarrow \mathrm{names}(\mathrm{Ran}(E' \downarrow N'))) \tag{6.6}$$

Now, from (6.5), from (6.6), and from the definitions of strong enrichment and restriction, we have $B \sqsupseteq (B' \Downarrow N')$, as required. □

The following proposition states that, provided B_1 and B_2 both are the identity with respect to restriction to some name set N, if B strongly enriches both B_1 and B_2 then B_1 equals B_2.

Proposition 6.3.2 *If* $B \sqsupseteq B_1$ *and* $B \sqsupseteq B_2$ *and* $B_1 = B_1 \Downarrow N$ *and* $B_2 = B_2 \Downarrow N$ *then* $B_1 = B_2$.

PROOF The proof is by induction on the structure of bases.

CASE $B = E$ For this case, the result follows directly from the definitions of strong enrichment and restriction on environments.

CASE $B = E.B'$ It follows from assumptions and from the definitions of strong enrichment and restriction that there exist environments E_1 and E_2 and bases B_1' and B_2' such that $B_1 = E_1.B_1'$ and $B_2 = E_2.B_2'$ and

$$E \sqsupseteq E_1 \tag{6.7}$$
$$E \sqsupseteq E_2 \tag{6.8}$$
$$B' \sqsupseteq B_1' \tag{6.9}$$
$$B' \sqsupseteq B_2' \tag{6.10}$$
$$E_1 = E_1 \downarrow N \tag{6.11}$$
$$E_2 = E_2 \downarrow N \tag{6.12}$$
$$B_1' = B_1' \Downarrow (\mathrm{names}(\mathrm{Ran}(E_1 \downarrow N))) \tag{6.13}$$
$$B_2' = B_2' \Downarrow (\mathrm{names}(\mathrm{Ran}(E_2 \downarrow N))) \tag{6.14}$$

From the definitions of strong enrichment and restriction on environments and from (6.7), (6.8), (6.11), and (6.12), we have

$$E_1 = E_2 \tag{6.15}$$

Moreover, we can apply induction to (6.9), (6.10), (6.13), and (6.14) to get $B_1' = B_2'$, thus, from (6.15), we have $B_1 = B_2$, as required. □

We now show that if some basis B strongly enriches another basis B_0 and if B_0 is the identity with respect to restriction to some name set N then B_0 equals B restricted to N.

Proposition 6.3.3 *If* $B \sqsupseteq B_0$ *and* $B_0 = B_0 \Downarrow N$ *then* $B_0 = B \Downarrow N$.

PROOF The proof is by induction on the structure of bases.

CASE $B = E$ For this case the result follows directly from the definitions of strong enrichment and restriction.

CASE $B = E.B'$ It follows from assumptions and from the definitions of strong enrichment and restriction that there exist an environment E_0 and a basis B'_0 such that $B_0 = E_0.B'_0$ and

$$E \sqsupseteq E_0 \tag{6.16}$$
$$B' \sqsupseteq B'_0 \tag{6.17}$$
$$E_0 = E_0 \downarrow N \tag{6.18}$$
$$B'_0 = B'_0 \Downarrow (\text{names}(\text{Ran}(E_0 \downarrow N))) \tag{6.19}$$

From the definitions of strong enrichment and restriction on environments and from (6.16) and (6.18), we have

$$E_0 = E \downarrow N \tag{6.20}$$

Now, from the definition of restriction on environments and from (6.19) and (6.20), we have

$$B'_0 = B'_0 \Downarrow (\text{names}(\text{Ran}(E \downarrow N))) \tag{6.21}$$

We can now apply induction to (6.17) and (6.21) to get

$$B'_0 = B' \Downarrow (\text{names}(\text{Ran}(E \downarrow N))) \tag{6.22}$$

From the definition of restriction and from (6.20) and (6.22), we have $B_0 = B \Downarrow N$, as required. □

6.4 Compilation

Compilation is defined in terms of translation steps. The rules for compilation allow inferences among sentences of the form

$$B \vdash p \Rightarrow (N)(B', p')$$

where B and B' are bases, p is a source program unit, p' is a target program unit, and N is the set of names that are generated during compilation. Sentences of this form are read "p compiles to $(N)(B', p')$ in B." Again, the prefix (N) in the object $(N)(B', p')$ binds names and we consider objects of this form equal up to renaming of bound names. We refer to bases in objects of the form $(N)(B, p)$ as *export* bases.

Program units $\boxed{B \vdash p \Rightarrow (N)(B', p')}$

$$\frac{E \vdash p \Rightarrow (N)(E', p')}{E \vdash p \Rightarrow (N)(E', p')} \tag{6.23}$$

$$\frac{\begin{array}{cc} E \vdash p \Rightarrow (N)(E', p') & (N \cup N') \cap \mathrm{names}(E.B) = \emptyset \\ B \vdash p' \Rightarrow (N')(B', p'') & N' \cap (N \cup \mathrm{names}(E', p')) = \emptyset \end{array}}{E.B \vdash p \Rightarrow (N \cup N')(E'.B', p'')} \tag{6.24}$$

Comment:

(6.24) The side conditions in this rule ensures that generated names are unique.

The following proposition states that compilation of a program unit depends only on the part of the basis that describes names that are used in the program unit. The usage propagation property of translation steps (Property 6.2.2) is essential for this proposition.

Proposition 6.4.1 (Compilation is closed under strong enrichment) *If* $B \vdash p \Rightarrow (N)(B', p')$ *and* $B'' \sqsupseteq (B \Downarrow \mathrm{uses}\ p)$ *then* $B'' \vdash p \Rightarrow (N)(B', p')$.

PROOF The proof is by induction over the structure of bases.

CASE $B = E$ The result follows from assumptions, from rule 6.23, and from Property 6.2.1.

CASE $B = E.B_1$ From assumptions and from rule 6.24, we have

$$E \vdash p \Rightarrow (N'')(E', p') \tag{6.25}$$
$$(N'' \cup N') \cap \text{names } B = \emptyset \tag{6.26}$$
$$B_1 \vdash p' \Rightarrow (N')(B''', p'') \tag{6.27}$$
$$N' \cap (N'' \cup \text{names}(E', p')) = \emptyset \tag{6.28}$$
$$N = N'' \cup N' \quad \text{and} \quad B' = E'.B''' \tag{6.29}$$

By appropriate renaming of bound names of $(N'' \cup N')(E'.B''', p'')$, we can assume

$$(N'' \cup N') \cap \text{names } B'' = \emptyset \tag{6.30}$$

Write B'' in the form $E''.B_1''$. From assumptions and from the definitions of strong enrichment and restriction, we have

$$E'' \sqsupseteq (E \downarrow \text{uses } p) \tag{6.31}$$
$$B_1'' \sqsupseteq (B_1 \Downarrow \text{names}(\text{Ran}(E \downarrow \text{uses } p))) \tag{6.32}$$

It follows from (6.31), from (6.25), and from Property 6.2.1 that we have

$$E'' \vdash p \Rightarrow (N'')(E', p') \tag{6.33}$$

Moreover, from (6.25) and from Property 6.2.2, we have

$$\text{uses } p' \subseteq \text{names}(\text{Ran}(E \downarrow \text{uses } p)) \tag{6.34}$$

Now, from (6.32), from (6.34), and from Proposition 6.3.1, we have

$$B_1'' \sqsupseteq (B_1 \Downarrow \text{uses } p') \tag{6.35}$$

We can now apply induction to (6.27) and (6.35) to get

$$B_1'' \vdash p' \Rightarrow (N')(B''', p'') \tag{6.36}$$

From rule 6.24 and from (6.33), (6.29), (6.30), (6.36), and (6.28), we have $B'' \vdash p \Rightarrow (N)(B', p'')$, as required. □

We now demonstrate that compilation is deterministic:

Proposition 6.4.2 (Compilation is deterministic) *If* $B \vdash p \Rightarrow (N_1)(B_1, p_1)$ *and* $B \vdash p \Rightarrow (N_2)(B_2, p_2)$ *then* $(N_1)(B_1, p_1) = (N_2)(B_2, p_2)$.

PROOF The proof is by induction on the structure of bases.

CASE $B = E$ From Property 6.2.3, from assumptions, and from rule 6.23, we have $(N_1)(B_1, p_1) = (N_2)(B_2, p_2)$, as required.

CASE $B = E.B'$ Write B_1 in the form $E_1.B'_1$ and write B_2 in the form $E_2.B'_2$. From assumptions and from rule 6.24, we have

$$E \vdash p \Rightarrow (N_1)(E_1, p_1) \tag{6.37}$$
$$E \vdash p \Rightarrow (N_2)(E_2, p_2) \tag{6.38}$$
$$B' \vdash p_1 \Rightarrow (N'_1)(B'_1, c_1) \tag{6.39}$$
$$B' \vdash p_2 \Rightarrow (N'_2)(B'_2, c_2) \tag{6.40}$$
$$B \vdash p \Rightarrow (N_1 \cup N'_1)(B_1, c_1) \tag{6.41}$$
$$B \vdash p \Rightarrow (N_2 \cup N'_2)(B_2, c_2) \tag{6.42}$$

From Property 6.2.3 and from (6.37) and (6.38), we have $(N_1)(E_1, p_1) = (N_2)(E_2, p_2)$, thus, we have

$$E_1 = E_2 \quad \text{and} \quad p_1 = p_2 \tag{6.43}$$

for some choice of N_1 and N_2 such that $N_1 = N_2$. Moreover, by applying induction to (6.39) and (6.40), we have $(N'_1)(B'_1, c_1) = (N'_2)(B'_2, c_2)$, thus, we have

$$B'_1 = B'_2 \quad \text{and} \quad c_1 = c_2 \tag{6.44}$$

for some choice of N'_1 and N'_2 such that $N'_1 = N'_2$. It now follows from (6.43) and (6.44) that we have $(N_1 \cup N'_1)(B_1, c_1) = (N_2 \cup N'_2)(B_2, c_2)$, as required. □

6.5 A Framework for Separate Compilation

Based on the notion of compilation that we developed in the previous section, we now present a simple framework for managing separate compilation or

what we call cut-off incremental recompilation. Recall that the rules for compilation allow inferences among sentences of the form

$$B \vdash p \Rightarrow (N)(B', c)$$

where B is a basis, p is a source program unit, N is a set of names, and c is a target program unit.

A *project* prj is a sequence of pairs of a program unit identifier (file name) and a program unit:

prj	::=	$pid \triangleright p$	program unit
	|	$prj_1\ prj_2$	sequence
	|	ε	empty

Projects are entities provided by the programmer. In other separate compilation schemes a make file or a configuration file provides similar information.

Target program units may be put together in a sequence to form *code objects*. We use o to range over code object:

o	::=	c	target program unit
	|	$o_1\ ;\ o_2$	sequence
	|	ε	empty

When two code objects o_1 and o_2 are put together to form the code object $o = o_1\ ;\ o_2$, the code objects o_1 and o_2 are implicitly linked in the sense that used names of o_2 may be bound by declared names of o_1. The set of declared names of o is the union of the declared names of o_1 and o_2. Moreover, the set of used names of o is the union of used names of o_1 and the subtraction of the used names of o_2 with the declared names of o_1.

We now define the notion of a repository. Informally, a repository holds information about compiled program units. A *repository* R is a finite map from program unit identifiers to objects of the form $(B, p, (N)(B', c))$, where B and B' are bases, p is a program unit, N is a set of names, and c is a target program unit. The basis B is referred to as the *import* basis of the repository entry and holds assumptions for compiling p. The object $(N)(B', c)$ is the result of previously compiling p in a basis that strongly enriches B. We shall refer to B' as the *export* basis of the repository entry. A repository R is *well-formed*, written $\vdash R$, if for all objects $(B, p, (N)(B', c))$ in the range of R, we have $B \vdash p \Rightarrow (N)(B', c)$ and $B = B \Downarrow \text{uses } p$.

We now give rules for managing (or compiling) projects. The rules allow inferences among sentences of the form

$$R, B \vdash prj \Rightarrow (N)(B', o), R'$$

where R and R' are repositories, B and B' are bases, prj is a project, N is the set of names that are generated during compilation, and o is a code object. Sentences of this form are read "prj compiles to $(N)(B', o)$ and R' in (R, B)."

Projects $\boxed{R, B \vdash prj \Rightarrow (N)(B', o), R'}$

$$\frac{}{R, B \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \varepsilon), \{\}} \tag{6.45}$$

$$\frac{\begin{array}{c} R(pid) = (B_0, p', (N)(B', c)) \quad p = p' \quad B \sqsupseteq B_0 \\ N \cap \text{names } B = \emptyset \quad R' = \{pid \mapsto R(pid)\} \end{array}}{R, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R'} \tag{6.46}$$

$$\frac{\begin{array}{c} (pid \notin \text{Dom } R) \vee (R(pid) = (B_0, p', A) \wedge (p \neq p' \vee B \not\sqsupseteq B_0)) \\ B \vdash p \Rightarrow (N)(B', c) \quad N \cap \text{names } B = \emptyset \\ R' = \{pid \mapsto (B \Downarrow \text{uses } p, p, (N)(B', c))\} \end{array}}{R, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R'} \tag{6.47}$$

$$\frac{\begin{array}{c} R, B \vdash prj_1 \Rightarrow (N_1)(B_1, o_1), R_1 \\ R, B + B_1 \vdash prj_2 \Rightarrow (N_2)(B_2, o_2), R_2 \\ (N_1 \cup N_2) \cap \text{names } B = \emptyset \quad N_2 \cap (N_1 \cup \text{names}(B_1, o_1)) = \emptyset \end{array}}{R, B \vdash prj_1 \ prj_2 \Rightarrow (N_1 \cup N_2)(B_1 + B_2, o_1 \ ; \ o_2), R_1 + R_2} \tag{6.48}$$

Comments:

(6.46) This rule allows for a compilation result in the repository for pid to be reused if certain side conditions hold. First, the program unit must

not have changed since the result was stored in the repository. This requirement is expressed in the rule with the side condition $p = p'$; in an implementation, file modification dates or cryptographic checksums may be used to check for this requirement. Second, the basis in which the project is compiled must strongly enrich the import basis of the repository entry for the program unit. Finally, the generated names of the object found in the repository must be fresh with respect to the compilation basis in which the project is compiled.

(6.47) This rule corresponds to compilation. If the side condition in this rule is satisfied then there is no object in the repository that can be reused; thus, the program unit must be (re)compiled. It is never the case that both rule 6.46 and rule 6.47 are applicable, given R, B, and prj.

(6.48) The side conditions in this rule ensure that generated names are unique.

The rules for managing projects are non-deterministic because the choice of the name set N_1 in rule 6.48 has influence on whether rule 6.46 is applicable for program units in prj_2. This non-determinism, however, has no influence on the correctness result that we shall demonstrate in Section 6.6. However, this flexibility in the rules is exactly what allows for cut-off recompilation. We shall return to this issue in Section 6.7.

6.6 Correctness of the Framework

Correctness of the separate compilation framework expresses that the result obtained by compiling a project in some well-formed repository can be obtained by compiling the project in any well-formed repository. First, we demonstrate that compilation of a project in some well-formed repository results in a well-formed repository:

Proposition 6.6.1 (Well-formed repositories) *If* $\vdash R$ *and* $R, B \vdash prj \Rightarrow (N)(B', o), R'$ *then* $\vdash R'$.

PROOF The proof is by induction on the derivation of $R, B \vdash prj \Rightarrow (N)(B', o), R'$.

CASE rule 6.45 In this case we have $R' = \{\}$, which is well-formed, as required.

CASE rule 6.46 From assumptions, we have $\vdash R'$, as required.

CASE rule 6.47 Let $B'' = B \Downarrow \text{uses } p$. From assumptions and from rule 6.47, we have

$$B \vdash p \Rightarrow (N)(B', c) \tag{6.49}$$

$$R' = \{pid \mapsto (B'', p, (N)(B', c))\} \tag{6.50}$$

From reflexivity of strong enrichment, we have

$$B'' \sqsupseteq (B \Downarrow \text{uses } p) \tag{6.51}$$

Now, from Proposition 6.4.1 and from (6.49), (6.50), and (6.51), we have $B'' \vdash p \Rightarrow (N)(B', c)$. Moreover, because $B'' = B \Downarrow \text{uses } p$, we have $B'' = B'' \Downarrow \text{uses } p$. Thus, from the definition of well-formedness of repositories and from (6.50), we have $\vdash R'$, as required.

CASE rule 6.48 From assumptions, from rule 6.48 and by applying induction twice, we have $\vdash R_1$ and $\vdash R_2$. From the definition of well-formedness of repositories, we have $\vdash (R_1 + R_2)$, as required. □

Correctness of the separate compilation scheme is expressed by the following proposition:

Proposition 6.6.2 (Correctness) *If* $\vdash R_1$ *and* $\vdash R_2$ *and* $R_1, B \vdash prj \Rightarrow (N)(B', o), R$ *then* $R_2, B \vdash prj \Rightarrow (N)(B', o), R$.

PROOF The proof is by induction on the derivation of $R_1, B \vdash prj \Rightarrow (N)(B', o), R$.

CASE rule 6.45 The result follows from assumptions and from rule 6.45.

CASE rule 6.46 From assumptions and from rule 6.46, we have

$$\vdash R_1 \tag{6.52}$$

$$\vdash R_2 \tag{6.53}$$

$$R_1, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R \tag{6.54}$$

$$R_1(pid) = (B_0, p, (N)(B', c)) \tag{6.55}$$

$$B \sqsupseteq B_0 \tag{6.56}$$

$$N \cap \text{names } B = \emptyset \tag{6.57}$$

$$R = \{pid \mapsto R_1(pid)\} \tag{6.58}$$

From the definition of well-formed repositories and from (6.52) and (6.55), we have

$$B_0 \vdash p \Rightarrow (N)(B', c) \tag{6.59}$$
$$B_0 = B_0 \Downarrow (\text{uses } p) \tag{6.60}$$

There are now two sub-cases to consider. First, consider the case where rule 6.46 is applicable. Then, there exists a basis B_0' such that

$$R_2(pid) = (B_0', p, (N)(B', c)) \tag{6.61}$$
$$B \sqsupseteq B_0' \tag{6.62}$$

From the definition of well-formed repositories and from (6.53) and (6.61), we have

$$B_0' = B_0' \Downarrow (\text{uses } p) \tag{6.63}$$

It follows from Proposition 6.3.2 and from (6.56), (6.62), (6.60), and (6.63) that we have $B_0' = B_0$. Now, from (6.55), (6.61), we have $R_1(pid) = R_2(pid)$, hence, from rule 6.46 and from (6.61), (6.62), (6.57), and (6.58), we have $R_2, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R$, as required.

Second, consider the case where rule 6.47 is applicable. Then, there exist a basis B_0', a program unit p', and an object A such that

$$(pid \notin \text{Dom } R_2) \vee (R_2(pid) = (B_0', p', A) \wedge (p \neq p' \vee B \not\sqsupseteq B_0')) \tag{6.64}$$

From (6.56) and from (6.60), we have

$$B \sqsupseteq B_0 \Downarrow (\text{uses } p) \tag{6.65}$$

Now, from Proposition 6.4.1 and from (6.65) and (6.59), we have

$$B \vdash p \Rightarrow (N)(B', c) \tag{6.66}$$

Moreover, from Proposition 6.3.3 and from (6.56) and (6.60), we have $B_0 = B \Downarrow (\text{uses } p)$, hence, from rule 6.48 and from (6.64), (6.66), (6.57), (6.58), and (6.55), we have $R_2, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R$, as required.

CASE rule 6.47 From assumptions and from rule 6.47, we have

$$\vdash R_1 \tag{6.67}$$
$$\vdash R_2 \tag{6.68}$$
$$(pid \notin \text{Dom } R_1) \vee (R_1(pid) = (B_0', p', A) \wedge (p \neq p' \vee B \not\sqsupseteq B_0')) \tag{6.69}$$
$$B \vdash p \Rightarrow (N)(B', c) \tag{6.70}$$
$$N \cap \text{names } B = \emptyset \tag{6.71}$$
$$R = \{pid \mapsto (B \Downarrow \text{uses } p, p, (N)(B', c))\} \tag{6.72}$$
$$R_1, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R \tag{6.73}$$

There are now two sub-cases to consider. First, consider the case where rule 6.47 is applicable. Then, there exist a basis B_0'', a program unit p'', and an object A' such that

$$(pid \notin \text{Dom } R_2) \vee (R_2(pid) = (B_0'', p'', A') \wedge (p \neq p'' \vee B \not\sqsupseteq B_0'')) \tag{6.74}$$

It follows from rule 6.47 and from (6.74), (6.70), (6.71), and (6.72) that we have $R_2, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R$, as required.

Second, consider the case where rule 6.46 is applicable. Then, we have that there exist bases B_0 and B'', a name set N', and a target program unit c' such that

$$R_2(pid) = (B_0, p, (N')(B'', c')) \tag{6.75}$$
$$B \sqsupseteq B_0 \tag{6.76}$$

From the definition of well-formed repositories and from (6.68), we have

$$B_0 \vdash p \Rightarrow (N')(B'', c') \tag{6.77}$$
$$B_0 = B_0 \Downarrow (\text{uses } p) \tag{6.78}$$

Now, from (6.78) and from (6.76), we have $B \sqsupseteq (B_0 \Downarrow (\text{uses } p))$, thus, from Proposition 6.4.1 and from (6.77), we have

$$B \vdash p \Rightarrow (N')(B'', c') \tag{6.79}$$

Moreover, from Proposition 6.4.2 and from (6.77) and (6.79), we have

$$(N')(B'', c') = (N)(B', c) \tag{6.80}$$

From Proposition 6.3.3 and from (6.76) and (6.78), we have $B_0 = B \Downarrow (\text{uses } p)$, thus, from (6.75), (6.72), and (6.80), we have

$$R = \{pid \mapsto R_2(pid)\} \tag{6.81}$$

Now, from rule 6.46 and from (6.75), (6.80), (6.76), (6.71), and (6.81), we have $R_2, B \vdash pid \triangleright p \Rightarrow (N)(B', c), R$, as required.

CASE rule 6.48 From assumptions and from rule 6.48, we have

$$\vdash R_1 \tag{6.82}$$
$$\vdash R_2 \tag{6.83}$$
$$R_1, B \vdash prj_1 \Rightarrow (N_1)(B_1, o_1), R_1' \tag{6.84}$$
$$(N_1 \cup N_2) \cap \text{names } B = \emptyset \tag{6.85}$$
$$R_1, B + B_1 \vdash prj_2 \Rightarrow (N_2)(B_2, o_2), R_2' \tag{6.86}$$
$$N_2 \cap (N_1 \cup \text{names}(B_1, o_1)) = \emptyset \tag{6.87}$$
$$R_1, B \vdash prj_1\ prj_2 \Rightarrow (N_1 \cup N_2)(B_1 + B_2, o_1\ ;\ o_2), R_1' + R_2' \tag{6.88}$$

By applying induction twice to (6.82) and (6.84) and to (6.83) and (6.85), respectively, we get

$$R_2, B \vdash prj_1 \Rightarrow (N_1)(B_1, o_1), R_1' \tag{6.89}$$
$$R_2, B + B_1 \vdash prj_2 \Rightarrow (N_2)(B_2, o_2), R_2' \tag{6.90}$$

Now, from rule 6.48 and from (6.89), (6.85), (6.90), and (6.87), we have $R_2, B \vdash prj_1\ prj_2 \Rightarrow (N_1 \cup N_2)(B_1 + B_2, o_1\ ;\ o_2), R_1' + R_2'$, as required. □

Soundness of the separate compilation framework expresses that the result obtained by compiling a project in some well-formed repository also can be obtained by compiling the project in an empty repository. Soundness follows immediately from the more general correctness result.

Corollary 6.6.3 (Soundness) *If* $\vdash R$ *and* $R, B \vdash prj \Rightarrow (N)(B', o), R'$ *then* $\{\}, B \vdash prj \Rightarrow (N)(B', o), R'$.

Similarly, completeness of the separate compilation framework expresses that the result obtained by compiling a project in an empty repository also can be obtained by compiling the project in any well-formed repository. Completeness follows immediately from the more general correctness result.

Corollary 6.6.4 (Completeness) *If* $\vdash R$ *and* $\{\}, B \vdash prj \Rightarrow (N)(B', o), R'$ *then* $R, B \vdash prj \Rightarrow (N)(B', o), R'$.

6.7 Non-Determinism and Matching

As we mentioned in Section 6.5, the rules for managing projects are non-deterministic in the sense that the choice of the name set N_1 in rule 6.48 may influence whether rule 6.46 is applicable for program units in prj_2.

Almost all non-determinism may be eliminated by always generating fresh names during compilation and by allowing renaming of bound names only in rule 6.47 when the program unit p is compiled to the object $(N)(B', c)$. At this point, if an object is available in the repository for the program unit identifier *pid*, the goal is to choose the name set N such that the basis B' agrees, on as many entries as possible, with the export basis for *pid* in the repository. Call the export basis for *pid* in the repository for B. In an implementation, the process of renaming N can be done by *matching* the basis B' to agree with the basis B on as many entries as possible. Now, the reason it is not possible to eliminate the non-determinism completely is that different choices of the name set N can satisfy agreement for B and B' for different entries and thus may result in different repository objects being reused. As a simple example, assume a compiler basis is an environment that maps program variables, ranged over by `a` and `b`, to machine code labels, ranged over by l. Further, assume $B = \{\mathtt{a} \mapsto l_1, \mathtt{b} \mapsto l_2\}$, for some distinct machine code labels l_1 and l_2, and assume $(N)(B', c) = (\{l\})(\{\mathtt{a} \mapsto l, \mathtt{b} \mapsto l\}, c)$, for some c. There are now two possibilities for the choice of N, each of which satisfy agreement for either `a` or for `b`, exclusively.

The implementation of matching may be composed from matching functions for each translation step in a compiler.

Following the preceding strategy, the separate compilation framework may be implemented by use of standard linking technology; the framework is used in the ML Kit with Regions (aka the ML Kit Version 2) for providing cut-off incremental recompilation for the Standard ML Core language [TBE+97, Chapter 16]. In the next chapters, we shall see how the framework for cut-off incremental recompilation is extended to work in a setting where Standard ML Modules phrases are interpreted at compile time so as to propagate implementation details across Modules boundaries during compilation.

Chapter 7

The Language IntML

In Chapter 2, we presented the language ModML. In this chapter, we present an explicitly typed intermediate language that we shall use as the target language for a translation of ModML programs. The language is called IntML; it has no support for modules because, as we shall see in Chapter 8, module constructs are eliminated during the translation into IntML. The IntML language has support for let-polymorphism and it is explicitly typed so that only type checking (as opposed to type inference) is necessary to decide whether IntML declarations are well-typed. Later phases of a compiler may benefit from explicit type information in the intermediate language. Moreover, a compiler may sometimes wish to check that transformations on the intermediate language, such as in-lining and other optimisations, preserve typeability. This possibility has proved to be of practical importance for compiler development [TMC+96].

Variables in IntML are not identifiers stemming from ModML programs. Because ModML programs are flattened (e.g., module constructs are eliminated) when translated into IntML declarations, it is necessary to choose fresh variables, from a separate name space during translation, to ensure that identifiers stemming from ModML programs do not clash. The translation from ModML programs to IntML declarations maintains a mapping from ModML identifiers to IntML variables.

IntML does not have any type abstraction mechanism (besides let-polymorphism). That is, a type name can only be bound by a datatype declaration, which explicitly mentions the set of value constructors associated with the type name. For simplicity, IntML allows datatype declarations to have only one value constructor. Moreover, value constructors in IntML cannot

take arguments. It is straightforward, however, to extend IntML to allow for multiple value constructors for each type name and to allow value constructors to take arguments (see Section 7.5).

For the purpose of demonstrating type correctness of the translation from ModML programs to IntML declarations, IntML declarations may be composed (i.e., linked) to create a new IntML declaration. In an implementation, however, linking of program fragments can be performed at a much later stage (e.g., after machine code generation.)

In the sections to follow we present the syntax and the typing rules for IntML. In Section 7.3, we present the dynamic semantics for IntML in the style of a natural operational semantics. We demonstrate type soundness for IntML in Section 7.4; the proof is inspired by other proofs of type soundness [Tof88, Ler92] for Milners polymorphic type discipline [Mil78, DM82]. Leroy demonstrates [Ler92] that the techniques that we use to demonstrate soundness for IntML extends to other features of Standard ML including recursion and imperative constructs. In Section 7.5, we illustrate how datatypes in IntML can be extended to allow for multiple value constructors.

7.1 Syntax

We assume a denumerably infinite set IVar $\subseteq$ Name of IntML *variables* and a denumerably infinite set ICon $\subseteq$ Name of IntML *constructors*. We use x and c to range over variables and constructors, respectively. Further, we sometimes use a to denote either a variable or a constructor. IntML *typed expressions* and IntML *typed declarations* conform to the grammar in Figure 7.1.

The IntML language allows functions of the form $\lambda c : \tau.e$, where c is a constructor. Such a function resembles a case construct with only one branch. Dynamically, when applied, the function fails if it is not applied to a value denoting the constructor c. In Section 7.4, we demonstrate that for well-typed IntML program phrases, the pattern matching mechanism always succeeds. The constructor c is not bound within the body of the function, thus, the constructor occurs free in the function and may not be renamed.

By contrast, we consider the variable x in a function of the form $\lambda x : \tau.e$ to be bound within its body e. Similarly, we consider the type variables $\alpha^{(k)}$ in the value declaration

$$\texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e$$

e	::=	$\lambda a : \tau.e$	function
	|	$e_1\ e_2$	application
	|	a_τ	variable or constructor
	|	`let` d `in` e_2	local declaration
d	::=	`valdec` $x : \alpha^{(k)}.\tau$ `=` e	value declaration
	|	`datdec` $\alpha^{(k)} t$ `=` c	datatype declaration
	|	$d_1\ ;\ d_2$	sequence
	|	ε	empty

Figure 7.1: *Grammar for IntML typed expressions (e) and for IntML typed declarations (d).*

to be bound within τ and e. We consider functions and value declarations to be equivalent up to renaming of bound variables and bound type variables, respectively. The *declared* names of a declaration d, written $decl(d)$, is the set defined by the following equations:

$$\begin{aligned} decl(\texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e) &= \{x\} \\ decl(\texttt{datdec}\ \alpha^{(k)} t\ \texttt{=}\ c) &= \{t\} \\ decl(d_1\ ;\ d_2) &= decl(d_1) \cup decl(d_2) \\ decl(\varepsilon) &= \emptyset \end{aligned}$$

For declarations of the form $d_1\ ;\ d_2$, declared names of d_1 are bound in d_2. The IntML language supports local declarations through the use of expressions of the form `let` d `in` e. We consider such expressions equivalent up to renaming of declared names of d.

7.2 Typing Rules

Before we give the typing rules for IntML, we present the semantic objects that are involved. The semantic objects are those for the static semantics of ModML, presented in Section 2.3, including IntML variables, IntML constructors, and those semantic objects given in Figure 7.2.

The typing rules for expressions and declarations allow inferences among

$$\begin{array}{rcl} \Delta & \in & \text{IVarEnv} = \text{IVar} \xrightarrow{\text{fin}} \text{TypeScheme} \\ & & \text{IConEnv} = \text{ICon} \xrightarrow{\text{fin}} \text{TypeScheme} \\ \Theta & \in & \text{ITyEnv} = \text{TyName} \xrightarrow{\text{fin}} \text{IConEnv} \\ (\Theta, \Delta) \text{ or } \Gamma & \in & \text{IEnv} = \text{ITyEnv} \times \text{IVarEnv} \end{array}$$

Figure 7.2: *Additional semantic objects for IntML type checking.*

sentences of the forms

$$\Gamma \vdash e : \tau \quad \text{and} \quad \Gamma \vdash d : \Gamma'$$

where Γ and Γ' are typing environments, e is an expression, τ is a type, and d is a declaration. Sentences of the former form are read "e has type τ in Γ." Sentences of the latter form are read "d respects Γ' in Γ." When Γ is some typing environment, we write tyvars Γ to denote the set of type variables that occur free in Γ. Moreover, we write tynames Γ to denote the set of type names that occur free in Γ and names Γ to denote the set of names that occur free in Γ. Notice that a name or a type name occurs free in some object if it occurs free in the domain of some finite map within the object.

Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\tau = \tau^{(k)}t \quad \text{Dom}(\Gamma(t)) = \{c\} \quad \Gamma(t)(c) \succ \tau \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \lambda c : \tau.e : \tau \to \tau'} \tag{7.1}$$

$$\frac{\Gamma + \{x \mapsto \tau\} \vdash e : \tau' \quad x \notin \text{names } \Gamma}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'} \tag{7.2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \tag{7.3}$$

$$\frac{\Gamma(x) \succ \tau}{\Gamma \vdash x_\tau : \tau} \tag{7.4}$$

$$\frac{\tau = \tau^{(k)}t \quad \Gamma(t)(c) \succ \tau}{\Gamma \vdash c_\tau : \tau} \tag{7.5}$$

$$\frac{\Gamma \vdash d : \Gamma' \quad \Gamma + \Gamma' \vdash e : \tau \quad \text{Dom } \Gamma' \cap \text{tynames } \tau = \emptyset}{\Gamma \vdash \texttt{let}\ d\ \texttt{in}\ e : \tau} \tag{7.6}$$

Comments:

(7.1) The requirement $\text{Dom}(\Gamma(t)) = \{c\}$ ensures that the simple match mechanism is exhaustive.

(7.6) Locally declared type names must not escape a `let` expression.

Declarations $\boxed{\Gamma \vdash d : \Gamma'}$

$$\frac{x \notin \text{names } \Gamma \quad \Gamma \vdash e : \tau \quad \text{tyvars } \alpha^{(k)} \cap \text{tyvars } \Gamma = \emptyset}{\Gamma \vdash \texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e : \{x \mapsto \forall\alpha^{(k)}.\tau\}} \tag{7.7}$$

$$\frac{t \notin \text{names } \Gamma \quad \text{arity } t = k}{\Gamma \vdash \texttt{datdec}\ \alpha^{(k)}t\ \texttt{=}\ c : \{t \mapsto \{c \mapsto \forall\alpha^{(k)}.\alpha^{(k)}t\}\}} \tag{7.8}$$

$$\frac{\Gamma \vdash d_1 : \Gamma_1 \quad \Gamma + \Gamma_1 \vdash d_2 : \Gamma_2}{\Gamma \vdash d_1\ ;\ d_2 : \Gamma_1 + \Gamma_2} \tag{7.9}$$

$$\frac{}{\Gamma \vdash \varepsilon : \{\}} \tag{7.10}$$

Comment:

(7.8) An IntML value constructor need not be chosen distinct from IntML value constructors of other datatypes. The type name has to be fresh, however, formulated by the side condition $t \notin \text{names } \Gamma$.

The following proposition states that those names that are being declared by a well-typed IntML declaration are chosen fresh:

Proposition 7.2.1 (Freshness of declared names) *If* $\Gamma \vdash d : \Gamma'$ *then* $\text{Dom } \Gamma' \cap \text{names } \Gamma = \emptyset$ *and* $decl(d) = \text{Dom } \Gamma'$.

PROOF By induction over the structure of d. □

Recall from Section 2.6 on page 14 that when S is a substitution, we write tynames S to denote the type names that occur free in the range of S. The following proposition states that type checking is closed under substitution.

Proposition 7.2.2 (Type checking is closed under substitution) *Let* S *be a substitution. If* $\Gamma \vdash e : \tau$ *then* $S(\Gamma) \vdash S(e) : S(\tau)$. *Moreover, if* $\Gamma \vdash d : \Gamma'$ *and* $\text{Dom } \Gamma' \cap \text{tynames } S = \emptyset$ *then* $S(\Gamma) \vdash S(d) : S(\Gamma')$.

PROOF By induction over the structure of e and d.

CASE $e = x_\tau$ From assumptions and from rule 7.4, we have $\Gamma(x) \succ \tau$. Because generalisation is closed under substitution, we have $(S(\Gamma))(x) \succ S(\tau)$, thus, from rule 7.4, we have $S(\Gamma) \vdash S(e) : S(\tau)$, as required.

CASE $e = \texttt{let}\ d\ \texttt{in}\ e'$ From assumptions and from rule 7.6, we have

$$\Gamma \vdash d : \Gamma' \tag{7.11}$$

$$\Gamma + \Gamma' \vdash e' : \tau \tag{7.12}$$

$$\text{Dom } \Gamma' \cap \text{tynames } \tau = \emptyset \tag{7.13}$$

From Proposition 7.2.1 and from (7.11), we have

$$\text{Dom } \Gamma' \cap \text{tynames } \Gamma = \emptyset \tag{7.14}$$

$$decl(d) = \text{Dom } \Gamma' \tag{7.15}$$

From (7.13), (7.14), and (7.15), we can assume

$$\text{Dom } \Gamma' \cap \text{tynames } S = \emptyset \tag{7.16}$$

by appropriate renaming of locally declared names. From (7.16) and (7.13), we have

$$\text{Dom } \Gamma' \cap \text{tynames}(S(\tau)) = \emptyset \tag{7.17}$$

By applying induction to (7.11) and (7.16), we have

$$S(\Gamma) \vdash S(d) : S(\Gamma') \tag{7.18}$$

Moreover, by applying induction to (7.12), we have

$$S(\Gamma) + S(\Gamma') \vdash S(e') : S(\tau) \tag{7.19}$$

Now, from rule 7.6 and from (7.18), (7.19), and (7.17), we have $S(\Gamma) \vdash S(e) : S(\tau)$, as required.

CASE $d = \texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e$ From assumptions and from rule 7.7, we have

$$x \notin \text{names } \Gamma \tag{7.20}$$
$$\Gamma \vdash e : \tau \tag{7.21}$$
$$\text{tyvars } \alpha^{(k)} \cap \text{tyvars } \Gamma = \emptyset \tag{7.22}$$
$$\Gamma' = \{x \mapsto \forall\alpha^{(k)}.\tau\} \tag{7.23}$$
$$\text{Dom } \Gamma' \cap \text{tynames } S = \emptyset \tag{7.24}$$

From (7.20) and from (7.24), we have

$$x \notin \text{names}(S(\Gamma)) \tag{7.25}$$

By appropriate renaming of bound type variables, we can assume

$$\text{tyvars } \alpha^{(k)} \cap \text{Inv } S = \emptyset \tag{7.26}$$

From (7.22) and from (7.26), we have

$$\text{tyvars } \alpha^{(k)} \cap \text{tyvars}(S(\Gamma)) = \emptyset \tag{7.27}$$

Moreover, from (7.26), we have

$$S(\forall\alpha^{(k)}.\tau) = \forall\alpha^{(k)}.S(\tau) \tag{7.28}$$

By applying induction to (7.21), we have

$$S(\Gamma) \vdash S(e) : S(\tau) \tag{7.29}$$

It follows from rule 7.7 and from (7.25), (7.29), (7.27), (7.28), and (7.23) that we have $S(\Gamma) \vdash S(d) : S(\Gamma')$, as required.

CASE $d = \texttt{datdec}\ \alpha^{(k)}\ t\ \texttt{=}\ c$ From assumptions and from rule 7.8, we have

$$t \notin \text{names}\ \Gamma \tag{7.30}$$

$$\text{arity}\ t = k \tag{7.31}$$

$$\Gamma' = \{t \mapsto \{c \mapsto \forall\alpha^{(k)}.\alpha^{(k)}\ t\}\} \tag{7.32}$$

From assumptions and from (7.32), we have

$$t \notin \text{names}(S(\Gamma)) \tag{7.33}$$

Now, from rule 7.8 and from (7.33), (7.31), and (7.32), we have $S(\Gamma) \vdash S(d) : S(\Gamma')$, as required.

CASE $d = d_1\ ;\ d_2$ From assumptions and from rule 7.9, we have

$$\Gamma \vdash d_1 : \Gamma_1 \tag{7.34}$$

$$\Gamma + \Gamma_1 \vdash d_2 : \Gamma_2 \tag{7.35}$$

$$\Gamma' = \Gamma_1 + \Gamma_2 \tag{7.36}$$

From assumptions and from (7.36), we have

$$\text{Dom}\ \Gamma_1 \cap \text{tynames}\ S = \emptyset \tag{7.37}$$

$$\text{Dom}\ \Gamma_2 \cap \text{tynames}\ S = \emptyset \tag{7.38}$$

By applying induction to (7.34) and (7.37), we have

$$S(\Gamma) \vdash S(d_1) : S(\Gamma_1) \tag{7.39}$$

Moreover, by applying induction to (7.35) and (7.38), we have

$$S(\Gamma) + S(\Gamma_1) \vdash S(d_2) : S(\Gamma_2) \tag{7.40}$$

Now, from rule 7.9 and from (7.39), (7.40), and (7.36), we have $S(\Gamma) \vdash S(d) : S(\Gamma')$, as required.

The remaining cases follow similarly. □

7.3 Dynamic Semantics

Evaluation of an IntML typed phrase is defined by first erasing all type information from the phrase, thereby yielding an untyped phrase, and then evaluating this untyped phrase with respect to an environment that provides assumptions for those variables that occur free in the phrase.

IntML untyped phrases are defined by an erasure function on typed phrases. We use e_{u} and d_{u} to range over IntML untyped expressions and IntML untyped declarations, respectively. Moreover, when it is clear from context, we also use e and d to range over IntML untyped expressions and IntML untyped declarations, respectively. When p is some IntML typed phrase, the *erasure* of p, written $er(p)$, is defined by the following equations:

Untyped expressions $\boxed{er(e) = e_{\mathrm{u}}}$

$$\begin{aligned} er(\lambda a : \tau.e) &= \lambda a.er(e) \\ er(e_1\ e_2) &= er(e_1)\ er(e_2) \\ er(a_\tau) &= a \\ er(\texttt{let}\ d\ \texttt{in}\ e) &= \texttt{let}\ er(d)\ \texttt{in}\ er(e) \end{aligned}$$

Untyped Declarations $\boxed{er(d) = d_{\mathrm{u}}}$

$$\begin{aligned} er(\texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e) &= \texttt{valdec}\ x\ \texttt{=}\ er(e) \\ er(\texttt{datdec}\ \alpha^{(k)}t\ \texttt{=}\ c) &= \varepsilon \\ er(d_1\ ;\ d_2) &= er(d_1)\ ;\ er(d_2) \\ er(\varepsilon) &= \varepsilon \end{aligned}$$

Notice that untyped declarations do not include datatype declarations.

The semantics of untyped IntML phrases are given as a natural operational semantics. The rules are instrumented with extra rules for expressing that evaluation goes wrong if a non-function is used as a function in an application, if a variable is looked up in a dynamic environment but is not there, or if pattern matching fails.

We use IExp to denote the set of IntML untyped expressions. The semantic objects for the dynamic semantics are given in Figure 7.3. Constructors

$$
\begin{array}{rcl}
v & \in & \mathrm{Val} = \mathrm{ICon} \cup \mathrm{Clos} \\
\langle \lambda a.e, \mathcal{D} \rangle & \in & \mathrm{Clos} = (\mathrm{IVar} \cup \mathrm{ICon}) \times \mathrm{IExp} \times \mathrm{DynEnv} \\
\mathcal{D} & \in & \mathrm{DynEnv} = \mathrm{IVar} \xrightarrow{\text{fin}} \mathrm{Val} \\
r & \in & \mathrm{ExpResult} = \mathrm{Val} \cup \{\mathit{wrong}\} \\
\varrho & \in & \mathrm{DecResult} = \mathrm{DynEnv} \cup \{\mathit{wrong}\}
\end{array}
$$

Figure 7.3: Semantic objects for the dynamic semantics.

are values. The object *wrong* is not a value; it is the result of a faulty evaluation such as an attempt to apply a non-function to an argument.

The evaluation rules allow inferences among sentences of the forms

$$\mathcal{D} \vdash e \rightsquigarrow r \quad \text{and} \quad \mathcal{D} \vdash d \rightsquigarrow \varrho$$

where $\mathcal{D}$ is a dynamic environment, e is an IntML untyped expression, d is an IntML untyped declaration, r is an expression result, and ϱ is a declaration result. The former sentence is read "e evaluates to r in $\mathcal{D}$." The latter sentence is read "d evaluates to ϱ in $\mathcal{D}$."

Expressions $\boxed{\mathcal{D} \vdash e \rightsquigarrow r}$

$$\frac{}{\mathcal{D} \vdash \lambda a.e \rightsquigarrow \langle \lambda a.e, \mathcal{D} \rangle} \tag{7.41}$$

$$\frac{}{\mathcal{D} \vdash c \rightsquigarrow c} \tag{7.42}$$

$$\frac{\mathcal{D}(x) = v}{\mathcal{D} \vdash x \rightsquigarrow v} \tag{7.43}$$

$$\frac{x \notin \mathrm{Dom}\ \mathcal{D}}{\mathcal{D} \vdash x \rightsquigarrow \mathit{wrong}} \tag{7.44}$$

$$\frac{\mathcal{D} \vdash e_1 \rightsquigarrow \langle \lambda x.e, \mathcal{D}_0 \rangle \quad \mathcal{D} \vdash e_2 \rightsquigarrow v \quad \mathcal{D}_0 + \{x \mapsto v\} \vdash e \rightsquigarrow r}{\mathcal{D} \vdash e_1\ e_2 \rightsquigarrow r} \tag{7.45}$$

$$\frac{\mathcal{D} \vdash e_1 \rightsquigarrow \langle \lambda c.e, \mathcal{D}_0 \rangle \quad \mathcal{D} \vdash e_2 \rightsquigarrow c \quad \mathcal{D}_0 \vdash e \rightsquigarrow r}{\mathcal{D} \vdash e_1\ e_2 \rightsquigarrow r} \tag{7.46}$$

$$\frac{\mathcal{D} \vdash e_1 \rightsquigarrow \langle \lambda c.e, \mathcal{D}_0 \rangle \quad \mathcal{D} \vdash e_2 \rightsquigarrow r \quad c \neq r}{\mathcal{D} \vdash e_1\ e_2 \rightsquigarrow \mathit{wrong}} \tag{7.47}$$

$$\frac{\mathcal{D} \vdash e_1 \rightsquigarrow \mathit{wrong} \text{ or } c}{\mathcal{D} \vdash e_1\ e_2 \rightsquigarrow \mathit{wrong}} \tag{7.48}$$

$$\frac{\mathcal{D} \vdash e_1 \rightsquigarrow \langle \lambda x.e, \mathcal{D}_0 \rangle \quad \mathcal{D} \vdash e_2 \rightsquigarrow \mathit{wrong}}{\mathcal{D} \vdash e_1\ e_2 \rightsquigarrow \mathit{wrong}} \tag{7.49}$$

$$\frac{\mathcal{D} \vdash d \rightsquigarrow \mathcal{D}' \quad \mathcal{D} + \mathcal{D}' \vdash e \rightsquigarrow r}{\mathcal{D} \vdash \texttt{let}\ d\ \texttt{in}\ e \rightsquigarrow r} \tag{7.50}$$

$$\frac{\mathcal{D} \vdash d \rightsquigarrow \mathit{wrong}}{\mathcal{D} \vdash \texttt{let}\ d\ \texttt{in}\ e \rightsquigarrow \mathit{wrong}} \tag{7.51}$$

Comments:

(7.46) and (7.47) The simple pattern mechanism of IntML requires the constructor in the closure to be identical to the result of evaluating the argument of the application.

(7.48) The "or" in this rule is used to collapse what is really two rules into one.

Declarations $\boxed{\mathcal{D} \vdash d \rightsquigarrow \varrho}$

$$\frac{\mathcal{D} \vdash e \rightsquigarrow v}{\mathcal{D} \vdash \texttt{valdec}\ x\ \texttt{=}\ e \rightsquigarrow \{x \mapsto v\}} \tag{7.52}$$

$$\frac{\mathcal{D} \vdash e \rightsquigarrow \mathit{wrong}}{\mathcal{D} \vdash \texttt{valdec}\ x\ \texttt{=}\ e \rightsquigarrow \mathit{wrong}} \tag{7.53}$$

$$\frac{\mathcal{D} \vdash d_1 \rightsquigarrow \mathcal{D}_1 \quad \mathcal{D} + \mathcal{D}_1 \vdash d_2 \rightsquigarrow \mathcal{D}_2}{\mathcal{D} \vdash d_1\ ;\ d_2 \rightsquigarrow \mathcal{D}_1 + \mathcal{D}_2} \tag{7.54}$$

$$\frac{\mathcal{D} \vdash d_1 \rightsquigarrow \mathit{wrong}}{\mathcal{D} \vdash d_1\ ;\ d_2 \rightsquigarrow \mathit{wrong}} \tag{7.55}$$

$$\frac{\mathcal{D} \vdash d_1 \rightsquigarrow \mathcal{D}_1 \quad \mathcal{D} + \mathcal{D}_1 \vdash d_2 \rightsquigarrow \mathit{wrong}}{\mathcal{D} \vdash d_1\ ;\ d_2 \rightsquigarrow \mathit{wrong}} \tag{7.56}$$

$$\frac{}{\mathcal{D} \vdash \varepsilon \rightsquigarrow \{\}} \tag{7.57}$$

7.4 Type Soundness

In this section, we demonstrate type soundness for IntML. The result implies that a well-typed IntML program phrase does not evaluate to *wrong*. Thus, type soundness implies that evaluation of the program phrase will never attempt to apply a non-function to an argument and never look in vain for a variable in the dynamic environment. Moreover, type soundness of IntML guarantees that the simple pattern matching mechanism of IntML is exhaustive; that is, the set of constructors to match against is known at compile time.

To demonstrate that well-typed IntML program phrases do not go wrong, we first define a *consistency relation* that relates dynamic objects to static objects. The relation is written

$$\Theta \models A : B$$

where A is some dynamic object and where B is some static object. The type name environment Θ provides constructor environments for constructors in A. We define the relation by induction over the structure of A.

- $\Theta \models c : \tau^{(k)}t$ iff $\Theta(t)(c) \succ \tau^{(k)}t$
- $\Theta \models \langle \lambda a.e_{\mathrm{u}}, \mathcal{D} \rangle : \tau_1 \to \tau_2$ iff there exist a typed expression e and a typing environment Γ such that $\Theta \models \mathcal{D} : \Gamma$ and $\Gamma \vdash \lambda a : \tau_1.e : \tau_1 \to \tau_2$ and $er(e) = e_{\mathrm{u}}$
- $\Theta \models v : \sigma$ iff for all types τ such that $\sigma \succ \tau$, we have $\Theta \models v : \tau$
- $\Theta \models \mathcal{D} : (\Theta', \Delta)$ iff Dom $\mathcal{D}$ = Dom Δ and $\Theta \models \mathcal{D}(x) : \Delta(x)$ for all $x \in$ Dom $\mathcal{D}$ and Dom $\Theta \supseteq$ Dom Θ' and $\Theta(t) = \Theta'(t)$ for all $t \in$ Dom Θ'

Before we state the type soundness proposition, we shall prove some properties of the consistency relation. A type name environment is *closed* if it contains no free type variables. First, the consistency relation is closed under substitution:

Proposition 7.4.1 (Substitution) *Assume that Θ is closed. If $\Theta \models v : \tau$ then $\Theta \models v : S(\tau)$ for any substitution S.*

PROOF By induction over the structure of v.

CASE $v = c$ From assumptions and from the definition of the consistency relation, we have $\tau = \tau^{(k)}t$ and $\Theta(t)(c) \succ \tau$. Because generalisation is closed under substitution and because Θ is closed, we have $\Theta(t)(c) \succ S(\tau)$. It follows from the definition of the consistency relation that we have $\Theta \models c : S(\tau)$, as required.

CASE $v = \langle \lambda a.e_{\mathrm{u}}, \mathcal{D} \rangle$ From assumptions and from the definition of the consistency relation, we have $\tau = \tau_1 \to \tau_2$ for some types τ_1 and τ_2. Let Γ be a typing environment and let e be an expression such that $er(e) = e_{\mathrm{u}}$ and

$$\Theta \models \mathcal{D} : \Gamma \tag{7.58}$$

$$\Gamma \vdash \lambda a : \tau_1.e : \tau_1 \to \tau_2 \tag{7.59}$$

From Proposition 7.2.2 and from (7.59), we have

$$S(\Gamma) \vdash \lambda a : S(\tau_1).S(e) : S(\tau_1 \to \tau_2) \tag{7.60}$$

Let $x_0 \in \mathrm{Dom}\ \mathcal{D}$. Write $\Gamma(x_0)$ in the form $\forall\alpha^{(k)}.\tau_0$, with $\alpha^{(k)}$ chosen such that tyvars $\alpha^{(k)} \cap \mathrm{Inv}\ S = \emptyset$. We then have $S(\Gamma(x_0)) = \forall\alpha^{(k)}.S(\tau_0)$. Let $\tau^{(k)}$ be a sequence of types. Moreover, let S' be the substitution $\{\tau^{(k)}/\alpha^{(k)}\}$. From the definition of the consistency relation and from (7.58), we have

$$\Theta \models \mathcal{D}(x_0) : \tau_0 \tag{7.61}$$

Because Θ is closed, we can apply induction to (7.61) to get

$$\Theta \models \mathcal{D}(x_0) : (S' \circ S)(\tau_0) \tag{7.62}$$

From the definitions of the consistency relation and of generalisation and because (7.62) holds for any sequence $\tau^{(k)}$ of types, we have

$$\Theta \models \mathcal{D}(x_0) : S(\Gamma(x_0)) \tag{7.63}$$

Now, from the definition of the consistency relation and from (7.58) and because (7.63) holds for all $x_0 \in \mathrm{Dom}\ \mathcal{D}$, we have

$$\Theta \models \mathcal{D} : S(\Gamma) \tag{7.64}$$

It follows from the definition of the consistency relation and from (7.60) and (7.58) that we have $\Theta \models \langle \lambda a.e_{\mathrm{u}}, \mathcal{D}\rangle : S(\tau)$, as required. □

The consistency relation is closed under modification of type name environments:

Proposition 7.4.2 (Type name environment modification) *Assume that* $\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset$.

- *If* $\Theta \models v : \tau$ *then* $\Theta + \Theta' \models v : \tau$.
- *If* $\Theta \models v : \sigma$ *then* $\Theta + \Theta' \models v : \sigma$.
- *If* $\Theta \models \mathcal{D} : \Gamma$ *then* $\Theta + \Theta' \models \mathcal{D} : \Gamma$.

Proof By induction over the derivation of the consistency relation.

Case $\Theta \models v : \tau$ and $v = c$ From assumptions and from the definition of the consistency relation, we have $\tau = \tau^{(k)}t$ and $\Theta(t)(c) \succ \tau$, for some type name t and for some types $\tau^{(k)}$. Because $\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset$, we have

$(\Theta + \Theta')(t)(c) \succ \tau$, thus, from the definition of the consistency relation, we have $\Theta + \Theta' \models v : \tau$, as required.

CASE $\Theta \models v : \tau$ and $v = \langle \lambda a.e_{\mathrm{u}}, \mathcal{D} \rangle$ From assumptions and from the definition of the consistency relation, we have that there exist types τ_1 and τ_2, an expression e, and a typing environment Γ, such that

$$\tau = \tau_1 \to \tau_2 \tag{7.65}$$
$$\Theta \models \mathcal{D} : \Gamma \tag{7.66}$$
$$\Gamma \vdash \lambda a : \tau_1.e : \tau \tag{7.67}$$
$$er(e) = e_{\mathrm{u}} \tag{7.68}$$
$$\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset \tag{7.69}$$

By applying induction to (7.66) and (7.69), we have

$$\Theta + \Theta' \models \mathcal{D} : \Gamma \tag{7.70}$$

From the definition of the consistency relation and from (7.65), (7.67), (7.68), and (7.70), we have $\Theta + \Theta' \models v : \tau$, as required.

CASE $\Theta \models v : \sigma$ Let τ be some type such that $\sigma \succ \tau$. From assumptions and from the definition of the consistency relation, we have

$$\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset \tag{7.71}$$
$$\Theta \models v : \tau \tag{7.72}$$

By applying induction to (7.71) and (7.72), we have

$$\Theta + \Theta' \models v : \tau \tag{7.73}$$

Because (7.73) holds for any type τ such that $\sigma \succ \tau$, we have from the definition of the consistency relation that $\Theta + \Theta' \models v : \sigma$, as required.

CASE $\Theta \models \mathcal{D} : \Gamma$ From assumptions and from the definition of the consistency relation, we have

$$\mathrm{Dom}\ \mathcal{D} = \mathrm{Dom}(\Delta \text{ of } \Gamma) \tag{7.74}$$
$$\Theta \models \mathcal{D}(x) : \Gamma(x) \text{ for all } x \in \mathrm{Dom}\ \mathcal{D} \tag{7.75}$$
$$\mathrm{Dom}\ \Theta \supseteq \mathrm{Dom}(\Theta \text{ of } \Gamma) \tag{7.76}$$
$$\Theta(t) = \Gamma(t) \text{ for all } t \in \mathrm{Dom}(\Theta \text{ of } \Gamma) \tag{7.77}$$
$$\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset \tag{7.78}$$

By applying induction to (7.75) and (7.78) for each $x \in \mathrm{Dom}\ \mathcal{D}$, we have

$$\Theta + \Theta' \models \mathcal{D}(x) : \Gamma(x) \text{ for all } x \in \mathrm{Dom}\ \mathcal{D} \tag{7.79}$$

From (7.76), we have

$$\mathrm{Dom}(\Theta + \Theta') \supseteq \mathrm{Dom}(\Theta \text{ of } \Gamma) \tag{7.80}$$

Moreover, from (7.76), (7.77), and (7.78), we have

$$(\Theta + \Theta')(t) = \Gamma(t) \text{ for all } t \in \mathrm{Dom}(\Theta \text{ of } \Gamma) \tag{7.81}$$

It follows from the definition of the consistency relation and from (7.74), (7.79), (7.80), and (7.81) that we have $\Theta + \Theta' \models \mathcal{D} : \Gamma$, as required. □

The main proposition expresses consistency between the static semantics of IntML and the dynamic semantics of IntML. The proposition must deal with the possibility that a value constructor of a locally declared datatype may escape in a closure. Consider the IntML expression

$$\texttt{let datdec } t \texttt{ = } c \texttt{ in } \lambda x : t_0.(\lambda x_0 : t.x)\ c \texttt{ end}$$

which has type $t_0 \to t_0$ under empty assumptions. The expression evaluates, under empty assumptions, to a closure that mentions the value constructor c. Consequently, we must relate the value to the type $t_0 \to t_0$ under assumptions that relate the type name t to an environment that mentions the value constructor c. The type name environment Θ'' in the proposition deals with this possibility. Moreover, the type name set T is used to express that such locally declared type names may be chosen fresh.

Proposition 7.4.3 (Type soundness) *Let Θ be closed. Assume that $\Theta \models \mathcal{D} : \Gamma$. Moreover, let T be a finite set of type names.*

- *If $\Gamma \vdash e : \tau$ and $\mathcal{D} \vdash er(e) \rightsquigarrow r$ then $r \neq$ wrong and there exists Θ'' such that Θ'' is closed and $(T \cup \mathrm{Dom}\ \Theta) \cap \mathrm{Dom}\ \Theta'' = \emptyset$ and $\Theta + \Theta'' \models r : \tau$.*

- *If $\Gamma \vdash d : \Gamma'$ and $\mathcal{D} \vdash er(d) \rightsquigarrow \varrho$ and $\Theta' = \Theta$ of Γ' and $\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset$ then $\varrho \neq$ wrong and there exists Θ'' such that $(\Theta' + \Theta'')$ is closed and $(T \cup \mathrm{Dom}(\Theta + \Theta')) \cap \mathrm{Dom}\ \Theta'' = \emptyset$ and $\Theta + \Theta' + \Theta'' \models \varrho : \Gamma'$.*

PROOF The proof is by induction over the structure of e and d.

CASE $e = \lambda a : \tau_1.e'$ From assumptions and from rules 7.1, 7.2, and 7.41, we have $r \neq \mathit{wrong}$, as required. Moreover, we have that there exists τ_2 such that

$$\Gamma \vdash \lambda a : \tau_1.e' : \tau \tag{7.82}$$
$$\tau = \tau_1 \to \tau_2 \tag{7.83}$$
$$r = \langle \lambda a.e'_{\mathrm{u}}, \mathcal{D} \rangle \tag{7.84}$$
$$e'_{\mathrm{u}} = er(e') \tag{7.85}$$

It follows from the definition of the consistency relation and from (7.82), (7.83), (7.84), and (7.85), that we have $\Theta \models r : \tau$, as required.

CASE $e = e_1\ e_2$ From assumptions and from rule 7.3, we have

$$\Theta \text{ is closed} \tag{7.86}$$
$$\Theta \models \mathcal{D} : \Gamma \tag{7.87}$$
$$\Gamma \vdash e_1 : \tau_2 \to \tau \tag{7.88}$$
$$\Gamma \vdash e_2 : \tau_2 \tag{7.89}$$
$$\mathcal{D} \vdash er(e) \rightsquigarrow r \tag{7.90}$$

By inspection of the evaluation rules and from (7.90), we see that either rule 7.45, rule 7.46, rule 7.47, rule 7.48, or rule 7.49 is used; in all cases, there exists r_1 such that

$$\mathcal{D} \vdash er(e_1) \rightsquigarrow r_1 \tag{7.91}$$

By applying induction to (7.86), (7.87), (7.88), and (7.91), we have $r_1 \neq \mathit{wrong}$ and there exists Θ_1 such that

$$\Theta_1 \text{ is closed} \tag{7.92}$$
$$(T \cup \mathrm{Dom}\ \Theta) \cap \mathrm{Dom}\ \Theta_1 = \emptyset \tag{7.93}$$
$$\Theta + \Theta_1 \models r_1 : \tau_2 \to \tau \tag{7.94}$$

From the definition of the consistency relation and from (7.94), we have, there exist $\langle \lambda a.e_{\mathrm{u}}, \mathcal{D}_0 \rangle$ and Γ_0 and e_0 such that

$$r_1 = \langle \lambda a.e_{\mathrm{u}}, \mathcal{D}_0 \rangle \tag{7.95}$$
$$\Theta + \Theta_1 \models \mathcal{D}_0 : \Gamma_0 \tag{7.96}$$
$$\Gamma_0 \vdash \lambda a : \tau_2.e_0 : \tau_2 \to \tau \tag{7.97}$$
$$er(e_0) = e_{\mathrm{u}} \tag{7.98}$$

From Proposition 7.4.2 and from (7.93) and (7.87), we have

$$\Theta + \Theta_1 \models \mathcal{D} : \Gamma \tag{7.99}$$

Moreover, from (7.86) and (7.92), we have

$$(\Theta + \Theta_1) \text{ is closed} \tag{7.100}$$

There are now two cases to consider depending on whether a is a constructor or a variable.

We first consider the case where $a = c$ for some constructor c. In this case, we have from rule 7.1 and (7.97) that there exist $\tau^{(k)}$ and t such that

$$\tau_2 = \tau^{(k)} t \tag{7.101}$$
$$\mathrm{Dom}(\Gamma_0(t)) = \{c\} \tag{7.102}$$
$$\Gamma_0(t)(c) \succ \tau_2 \tag{7.103}$$
$$\Gamma_0 \vdash e_0 : \tau \tag{7.104}$$

By inspection of the evaluation rules and from (7.90), (7.91), and (7.95), we see that either rule 7.46 or rule 7.47 is used; in both cases, there exists r_2 such that

$$\mathcal{D} \vdash er(e_2) \rightsquigarrow r_2 \tag{7.105}$$

By applying induction to (7.100), (7.99), (7.89), and (7.105), we have $r_2 \neq \mathit{wrong}$ and there exists Θ_2 such that

$$\Theta_2 \text{ is closed} \tag{7.106}$$
$$(T \cup \mathrm{Dom}(\Theta + \Theta_1)) \cap \mathrm{Dom}\ \Theta_2 = \emptyset \tag{7.107}$$
$$\Theta + \Theta_1 + \Theta_2 \models r_2 : \tau_2 \tag{7.108}$$

From the definition of the consistency relation and from (7.108) and (7.101), we have that there exists c' such that $r_2 = c'$ and

$$(\Theta + \Theta_1 + \Theta_2)(t)(c') \succ \tau_2 \tag{7.109}$$

From the definition of the consistency relation and from (7.96) and (7.103), we have $(\Theta + \Theta_1)(t) = \Gamma_0(t)$, thus, from (7.109), (7.107), (7.102), and (7.103), we have

$$c = c' = r_2 \tag{7.110}$$

By inspection of the evaluation rules and from (7.90), (7.91), (7.105), and (7.110), we see that rule 7.46 is used, thus, we have from (7.98) that

$$\mathcal{D}_0 \vdash er(e_0) \rightsquigarrow r \tag{7.111}$$

By applying induction to (7.100), (7.96), (7.104), and (7.111), we have $r \neq$ *wrong* and there exists Θ' such that

$$\Theta' \text{ is closed} \tag{7.112}$$
$$(T \cup \text{Dom}(\Theta + \Theta_1)) \cap \text{Dom } \Theta' = \emptyset \tag{7.113}$$
$$\Theta + \Theta_1 + \Theta' \models r : \tau \tag{7.114}$$

It follows from (7.92) and (7.112) that we have $(\Theta_1 + \Theta')$ is closed, as required. Moreover, from (7.93) and (7.113), we have $(T \cup \text{Dom } \Theta) \cap \text{Dom}(\Theta_1 + \Theta') = \emptyset$, as required.

We now consider the case where $a = x$ for some variable x. In this case, we have from rule 7.2 and (7.97) that

$$x \notin \text{names } \Gamma_0 \tag{7.115}$$
$$\Gamma_0 + \{x \mapsto \tau_2\} \vdash e_0 : \tau \tag{7.116}$$

By inspection of the evaluation rules and from (7.90), (7.91), and (7.95), we see that either rule 7.45 or rule 7.49 is used; in both cases, there exists r_2 such that

$$\mathcal{D} \vdash er(e_2) \rightsquigarrow r_2 \tag{7.117}$$

By applying induction to (7.100), (7.99), (7.89), and (7.117), we have $r_2 \neq$ *wrong* and there exists Θ_2 such that

$$\Theta_2 \text{ is closed} \tag{7.118}$$
$$(T \cup \text{Dom}(\Theta + \Theta_1)) \cap \text{Dom } \Theta_2 = \emptyset \tag{7.119}$$
$$\Theta + \Theta_1 + \Theta_2 \models r_2 : \tau_2 \tag{7.120}$$

Because $r_2 \neq$ *wrong*, it follows that rule 7.45 is used, thus, we have

$$\mathcal{D}_0 + \{x \mapsto r_2\} \vdash er(e_0) \rightsquigarrow r \tag{7.121}$$

From Proposition 7.4.2 and from (7.119) and (7.96), we have

$$\Theta + \Theta_1 + \Theta_2 \models \mathcal{D}_0 : \Gamma_0 \tag{7.122}$$

From the definition of the consistency relation and from (7.122) and (7.120), we have

$$\Theta + \Theta_1 + \Theta_2 \models (\mathcal{D}_0 + \{x \mapsto r_2\}) : (\Gamma_0 + \{x \mapsto \tau_2\}) \tag{7.123}$$

From (7.100) and (7.118), we have

$$(\Theta + \Theta_1 + \Theta_2) \text{ is closed} \tag{7.124}$$

Now, by applying induction to (7.124), (7.123), (7.116), and (7.121), we have $r \neq \mathit{wrong}$ and there exists Θ' such that

$$\Theta' \text{ is closed} \tag{7.125}$$

$$(T \cup \mathrm{Dom}(\Theta + \Theta_1 + \Theta_2)) \cap \mathrm{Dom}\ \Theta' = \emptyset \tag{7.126}$$

$$\Theta + \Theta_1 + \Theta_2 + \Theta' \models r : \tau \tag{7.127}$$

It follows from (7.92), (7.118), and (7.125) that we have $(\Theta_1 + \Theta_2 + \Theta')$ is closed, as required. Moreover, from (7.93), (7.119), and (7.126), we have $(T \cup \mathrm{Dom}\ \Theta) \cap \mathrm{Dom}(\Theta_1 + \Theta_2 + \Theta') = \emptyset$, as required.

CASE $e = c_\tau$ From assumptions and from rules 7.5 and 7.42, we have $r \neq \mathit{wrong}$, as required. In fact, we have $r = c$ and

$$\tau = \tau^{(k)}t \tag{7.128}$$

$$\Gamma(t)(c) \succ \tau \tag{7.129}$$

From assumptions and from the definition of the consistency relation, we have $\Theta(t) = \Gamma(t)$, thus, from (7.128) and (7.129) and from the definition of the consistency relation, we have $\Theta \models c : \tau$, as required.

CASE $e = x_\tau$ From assumptions and from rule 7.4, we have

$$\Gamma(x) \succ \tau \tag{7.130}$$

$$\Theta \models \mathcal{D} : \Gamma \tag{7.131}$$

It follows from assumptions, from the definition of the consistency relation, and from (7.130) that we have $x \in \mathrm{Dom}\ \mathcal{D}$, thus, by inspection of the evaluation rules, we see that rule 7.43 is used. Hence, we have $r \neq \mathit{wrong}$, as required; in fact, $r = \mathcal{D}(x)$. From the definition of the consistency relation and from (7.131), we have

$$\Theta \models r : \Gamma(x) \tag{7.132}$$

It follows from the definition of the consistency relation and from (7.130) and (7.132) that we have $\Theta \models r : \tau$, as required.

CASE $e = \texttt{let}\ d\ \texttt{in}\ e'$ From assumptions and from rule 7.6, we have, there exists Γ' such that

$$\Gamma \vdash d : \Gamma' \tag{7.133}$$
$$\Gamma + \Gamma' \vdash e' : \tau \tag{7.134}$$
$$\mathrm{Dom}(\Theta \text{ of } \Gamma') \cap \text{tynames } \tau = \emptyset \tag{7.135}$$
$$\Theta \text{ is closed} \tag{7.136}$$
$$\Theta \models \mathcal{D} : \Gamma \tag{7.137}$$
$$\mathcal{D} \vdash er(e) \rightsquigarrow r \tag{7.138}$$

By inspection of the evaluation rules and from (7.138), we see that either rule 7.50 or rule 7.51 is used; in both cases, there exists ϱ such that

$$\mathcal{D} \vdash er(d) \rightsquigarrow \varrho \tag{7.139}$$

Let $\Theta' = \Theta$ of Γ'. From Proposition 7.2.1 and (7.133), we have

$$\mathrm{Dom}\ \Gamma' \cap \text{names } \Gamma = \emptyset \tag{7.140}$$

Now, by appropriate renaming of bound type names (using (7.135) and (7.140)), we can assume

$$(T \cup \mathrm{Dom}\ \Theta) \cap \mathrm{Dom}\ \Theta' = \emptyset \tag{7.141}$$

By applying induction to (7.136), (7.137), (7.133), (7.139), and (7.141), we have $\varrho \neq \mathit{wrong}$ and there exists Θ_1 such that

$$(\Theta' + \Theta_1) \text{ is closed} \tag{7.142}$$
$$(T \cup \mathrm{Dom}(\Theta + \Theta')) \cap \mathrm{Dom}\ \Theta_1 = \emptyset \tag{7.143}$$
$$\Theta + \Theta' + \Theta_1 \models \varrho : \Gamma' \tag{7.144}$$

From (7.141) and (7.143), we have

$$\mathrm{Dom}\ \Theta \cap \mathrm{Dom}(\Theta' + \Theta_1) = \emptyset \tag{7.145}$$

Now, from Proposition 7.4.2 and from (7.137) and (7.145), we have

$$\Theta + \Theta' + \Theta_1 \models \mathcal{D} : \Gamma \tag{7.146}$$

Now, from the definition of the consistency relation and from (7.146) and (7.144), we have

$$\Theta + \Theta' + \Theta_1 \models (\mathcal{D} + \varrho) : (\Gamma + \Gamma') \tag{7.147}$$

By inspection of the evaluation rules and from (7.138) and (7.139), we see that rule 7.50 is used, thus, we have

$$\mathcal{D} + \varrho \vdash er(e') \rightsquigarrow r \tag{7.148}$$

From (7.136) and from (7.142), we have

$$(\Theta + \Theta' + \Theta_1) \text{ is closed} \tag{7.149}$$

By applying induction to (7.149), (7.147), (7.134), and (7.148), we have $r \neq \mathit{wrong}$, as required, and there exists Θ_2 such that

$$\Theta_2 \text{ is closed} \tag{7.150}$$

$$(T \cup \mathrm{Dom}(\Theta + \Theta' + \Theta_1)) \cap \mathrm{Dom}\ \Theta_2 = \emptyset \tag{7.151}$$

$$\Theta + \Theta' + \Theta_1 + \Theta_2 \models r : \tau \tag{7.152}$$

It follows from (7.149) and (7.150) that we have $(\Theta' + \Theta_1 + \Theta_2)$ is closed, as required. Moreover, from (7.141), (7.143), and (7.151), we have $(T \cup \mathrm{Dom}\ \Theta) \cap \mathrm{Dom}(\Theta' + \Theta_1 + \Theta_2) = \emptyset$, as required.

CASE $d = \texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e$ From assumptions and from rule 7.7, we have

$$x \notin \mathrm{names}\ \Gamma \tag{7.153}$$

$$\sigma = \forall \alpha^{(k)}.\tau \tag{7.154}$$

$$\Gamma \vdash e : \tau \tag{7.155}$$

$$\mathrm{tyvars}\ \alpha^{(k)} \cap \mathrm{tyvars}\ \Gamma = \emptyset \tag{7.156}$$

$$\Gamma' = \{x \mapsto \sigma\} \tag{7.157}$$

$$\Theta \models \mathcal{D} : \Gamma \tag{7.158}$$

$$\mathcal{D} \vdash er(d) \rightsquigarrow \varrho \tag{7.159}$$

$$\Theta \text{ is closed} \tag{7.160}$$

By inspection of the evaluation rules and from (7.159), we have that there exists r such that

$$\mathcal{D} \vdash er(e) \rightsquigarrow r \tag{7.161}$$

By applying induction to (7.160), (7.158), (7.155), and (7.161), we have $r \neq \mathit{wrong}$ and there exists Θ' such that $(T \cup \mathrm{Dom}\ \Theta) \cap \mathrm{Dom}\ \Theta' = \emptyset$, as required, and

$$\Theta' \text{ is closed} \tag{7.162}$$

$$\Theta + \Theta' \models r : \tau \tag{7.163}$$

From Proposition 7.4.1 and from (7.160), (7.162), and (7.163), we have

$$\Theta + \Theta' \models r : S(\tau) \text{ for any substitution } S \tag{7.164}$$

From the definition of the consistency relation, from the definition of generalisation, and from (7.154), we have

$$\Theta + \Theta' \models r : \sigma \tag{7.165}$$

By inspection of the evaluation rules and from (7.159) and (7.161) and because $r \neq \mathit{wrong}$, we see that rule 7.52 is used, thus, $\varrho \neq \mathit{wrong}$, as required; in fact

$$\varrho = \{x \mapsto r\} \tag{7.166}$$

It follows from the definition of the consistency relation and from (7.165), (7.157), and (7.166) that we have $\Theta + \Theta' \models \varrho : \Gamma'$, as required.

CASE $d = \texttt{datdec}\ \alpha^{(k)} t\ \texttt{=}\ c$ From assumptions and from rule 7.8 and rule 7.57, we have $\varrho \neq \mathit{wrong}$, as required. Moreover, we have

$$\Gamma' = \{t \mapsto \{c \mapsto \forall \alpha^{(k)}.\alpha^{(k)} t\}\} \tag{7.167}$$

$$\varrho = \{\} \tag{7.168}$$

Let $\Theta' = \Theta$ of Γ'. From the definition of the consistency relation and from (7.167) and (7.168), we have Θ' is closed and $\Theta + \Theta' \models \varrho : \Gamma'$, as required.

CASE $d = d_1\ ;\ d_2$ From assumptions and from rule 7.9, we have

$$\Gamma \vdash d_1 : \Gamma_1 \tag{7.169}$$

$$\Gamma + \Gamma_1 \vdash d_2 : \Gamma_2 \tag{7.170}$$

$$\Gamma' = \Gamma_1 + \Gamma_2 \tag{7.171}$$

$$\mathcal{D} \vdash d \rightsquigarrow \varrho \tag{7.172}$$

$$\Theta \models \mathcal{D} : \Gamma \tag{7.173}$$

$$\Theta' = \Theta \text{ of } \Gamma' \tag{7.174}$$

$$\mathrm{Dom}\ \Theta' \cap \mathrm{Dom}\ \Theta = \emptyset \tag{7.175}$$

$$\Theta \text{ is closed} \tag{7.176}$$

By inspection of the evaluation rules, we see that either rule 7.54, rule 7.55, or rule 7.56 is used; in all cases, there exists ϱ_1 such that

$$\mathcal{D} \vdash d_1 \rightsquigarrow \varrho_1 \tag{7.177}$$

Let $\Theta_1 = \Theta$ of Γ_1. From (7.171), (7.174), and (7.175), we have

$$\text{Dom } \Theta_1 \cap \text{Dom } \Theta = \emptyset \tag{7.178}$$

Let $\Theta_2 = \Theta$ of Γ_2. By applying induction to (7.176), (7.173), (7.178), (7.169), and (7.177), we have $\varrho_1 \neq \mathit{wrong}$ and there exists Θ_1' such that

$$\Theta + \Theta_1 + \Theta_1' \models \varrho_1 : \Gamma_1 \tag{7.179}$$

$$(T \cup \text{Dom } \Theta_2 \cup \text{Dom}(\Theta + \Theta_1)) \cap \text{Dom } \Theta_1' = \emptyset \tag{7.180}$$

$$(\Theta_1 + \Theta_1') \text{ is closed} \tag{7.181}$$

From (7.176) and from (7.181), we have

$$(\Theta + \Theta_1 + \Theta_1') \text{ is closed} \tag{7.182}$$

Moreover, from (7.178) and from (7.180), we have $\text{Dom } \Theta \cap \text{Dom}(\Theta_1 + \Theta_1') = \emptyset$, thus, from Proposition 7.4.2 and from (7.173), we have

$$\Theta + \Theta_1 + \Theta_1' \models \mathcal{D} : \Gamma \tag{7.183}$$

Now, from the definition of the consistency relation and from (7.183) and (7.179), we have

$$\Theta + \Theta_1 + \Theta_1' \models (\mathcal{D} + \varrho_1) : (\Gamma + \Gamma_1) \tag{7.184}$$

From (7.171), (7.174), (7.175), we have $\text{Dom } \Theta_2 \cap \text{Dom } \Theta = \emptyset$. Moreover, from Proposition 7.2.1 and from (7.170), we have $\text{Dom } \Theta_2 \cap \text{Dom } \Theta_1 = \emptyset$. Thus, from (7.180), we have

$$\text{Dom } \Theta_2 \cap \text{Dom}(\Theta + \Theta_1 + \Theta_1') = \emptyset \tag{7.185}$$

By inspection of the evaluation rules and because $\varrho_1 \neq \mathit{wrong}$, we see that either rule 7.54 or rule 7.56 is used; in both cases, there exists ϱ_2 such that

$$\mathcal{D} + \varrho_1 \vdash d_2 \rightsquigarrow \varrho_2 \tag{7.186}$$

By applying induction to (7.182), (7.184), (7.170), (7.186), and (7.185), we have $\varrho_2 \neq \mathit{wrong}$ and there exists Θ'_2 such that

$$\Theta + \Theta_1 + \Theta'_1 + \Theta_2 + \Theta'_2 \models \varrho_2 : \Gamma_2 \tag{7.187}$$

$$(T \cup \mathrm{Dom}(\Theta + \Theta_1 + \Theta'_1 + \Theta_2)) \cap \mathrm{Dom}\ \Theta'_2 = \emptyset \tag{7.188}$$

$$(\Theta_2 + \Theta'_2) \text{ is closed} \tag{7.189}$$

By inspection of the evaluation rules, we see that rule 7.54 is used, thus, we have $\varrho \neq \mathit{wrong}$, as required; in fact, we have $\varrho = \varrho_1 + \varrho_2$.

From (7.185) and from (7.188), we have

$$\mathrm{Dom}(\Theta + \Theta_1 + \Theta'_1) \cap \mathrm{Dom}(\Theta_2 + \Theta'_2) = \emptyset \tag{7.190}$$

Now, from Proposition 7.4.2 and from (7.179), we have

$$\Theta + \Theta_1 + \Theta'_1 + \Theta_2 + \Theta'_2 \models \varrho_1 : \Gamma_1 \tag{7.191}$$

From the definition of the consistency relation and from (7.191) and (7.187), we have

$$\Theta + \Theta_1 + \Theta'_1 + \Theta_2 + \Theta'_2 \models (\varrho_1 + \varrho_2) : (\Gamma_1 + \Gamma_2)$$

as required. Moreover, from (7.181) and from (7.189), we have $(\Theta_1 + \Theta'_1 + \Theta_2 + \Theta'_2)$ is closed. Finally, from (7.180) and from (7.188), we have

$$(T \cup \mathrm{Dom}(\Theta + \Theta_1 + \Theta_2)) \cap \mathrm{Dom}(\Theta'_1 + \Theta'_2) = \emptyset$$

as required.

CASE $d = \varepsilon$ The required result follows from assumptions, from the definition of the consistency relation, from rule 7.10, and from rule 7.57. □

To summarise, the following corollary expresses that closed well-typed IntML declarations cannot go wrong; the corollary is a direct consequence of Proposition 7.4.3:

Corollary 7.4.4 (Weak type soundness) *If* $\{\} \vdash d : \Gamma$ *and* $\{\} \vdash er(d) \rightsquigarrow \varrho$ *then* $\varrho \neq \mathit{wrong}$.

7.5 Datatypes with More Value Constructors

In this section, we illustrate how datatypes in IntML can be extended to allow for multiple value constructors. First, datatype declarations are extended to take the form

$$\texttt{datdec}\ \alpha^{(k)}t\ \texttt{=}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n \qquad \text{where } n \geq 1$$

and function expressions are extended to either be of the form $\lambda x : \tau.e$ or of the form

$$\lambda c_1 : \tau.e_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n : \tau.e_n \qquad \text{where } n \geq 1$$

For both language constructs, we require as a syntactic restriction that the value constructors c_1 thorough c_n are distinct.

Second, the typing rules 7.8 and 7.1 are extended appropriately. Here is the new rule for datatype declarations:

$$\frac{t \notin \text{names}\ \Gamma \quad \text{arity}\ t = k \quad \sigma = \forall\alpha^{(k)}.\alpha^{(k)}t}{\Gamma \vdash \texttt{datdec}\ \alpha^{(k)}t\ \texttt{=}\ c_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n : \{t \mapsto \{c_1 \mapsto \sigma, \cdots, c_n \mapsto \sigma\}\}} \tag{7.192}$$

And here is the new rule corresponding to rule 7.1:

$$\frac{\begin{array}{c}\tau = \tau^{(k)}t \quad \text{Dom}(\Gamma(t)) = \{c_1, \cdots, c_n\} \\ \Gamma(t)(c_i) \succ \tau \quad \Gamma \vdash e_i : \tau' \quad i = 1..n\end{array}}{\Gamma \vdash \lambda c_1 : \tau.e_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n : \tau.e_n : \tau \to \tau'} \tag{7.193}$$

Each of the propositions demonstrated in Section 7.2 also holds when IntML are extended to allow for datatypes with multiple value constructors.

The dynamic semantics of IntML is extended to allow closures of the form

$$\langle \lambda c_1.e_1\ \texttt{|}\ \cdots\ \texttt{|}\ c_n.e_n, \mathcal{D}\rangle$$

where e_1 thorough e_n are IntML untyped expressions, c_1 thorough c_n are constructors, and $\mathcal{D}$ is a dynamic environment. The dynamic semantics is modified by substituting rules 7.41, 7.46, and 7.47 with the following four rules:

Expressions (Modified) $\boxed{\mathcal{D} \vdash e \rightsquigarrow r}$

$$\frac{}{\mathcal{D} \vdash \lambda x.e \rightsquigarrow \langle \lambda x.e, \mathcal{D}\rangle} \tag{7.194}$$

$$\frac{}{\mathcal{D} \vdash \lambda c_1.e_1 \mid \cdots \mid c_n.e_n \rightsquigarrow \langle \lambda c_1.e_1 \mid \cdots \mid c_n.e_n, \mathcal{D} \rangle} \tag{7.195}$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash e \rightsquigarrow \langle \lambda c_1.e_1 \mid \cdots \mid c_n.e_n, \mathcal{D}_0 \rangle \\ \mathcal{D} \vdash e' \rightsquigarrow c_i \quad \mathcal{D}_0 \vdash e_i \rightsquigarrow r \quad i \in \{1, \cdots, n\}\end{array}}{\mathcal{D} \vdash e\ e' \rightsquigarrow r} \tag{7.196}$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash e \rightsquigarrow \langle \lambda c_1.e_1 \mid \cdots \mid c_n.e_n, \mathcal{D}_0 \rangle \\ \mathcal{D} \vdash e' \rightsquigarrow r \quad r \notin \{c_1, \cdots, c_n\}\end{array}}{\mathcal{D} \vdash e\ e' \rightsquigarrow \mathit{wrong}} \tag{7.197}$$

Type soundness can now be proved for the modified IntML language by modifying the consistency relation of Section 7.4 to relate closures to function types as follows:

- $\Theta \models \langle \lambda x.e_{\mathrm{u}}, \mathcal{D} \rangle : \tau_1 \to \tau_2$ iff there exist a typed expression e and a typing environment Γ such that $\Theta \models \mathcal{D} : \Gamma$ and $\Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2$ and $er(e) = e_{\mathrm{u}}$
- $\Theta \models \langle \lambda c_1.e_{\mathrm{u}}^1 \mid \cdots \mid c_n.e_{\mathrm{u}}^n, \mathcal{D} \rangle : \tau_1 \to \tau_2$ iff there exist typed expressions e_1 thorough e_n and a typing environment Γ such that $\Theta \models \mathcal{D} : \Gamma$ and $\Gamma \vdash \lambda c_1 : \tau_1.e_1 \mid \cdots \mid c_n : \tau_1.e_n : \tau_1 \to \tau_2$ and $er(e_i) = e_{\mathrm{u}}^i$ for all $i \in \{1, \cdots, n\}$.

The interesting case to inspect in the proof of type soundness is the case for function application. Here the type name environment in the consistency relation captures the set of constructors associated with a type name, thereby ensuring that a function of the form

$$\lambda c_1.e_1 \mid \cdots \mid c_n.e_n$$

is not applied to values different than c_1 thorough c_n.

Chapter 8

Static Interpretation of Modules

In this chapter, we show how ModML programs that do not contain opaque signature constraints (see Chapter 5) can be interpreted at compile time and translated into IntML programs in such a way that IntML phrases are generated only for ModML Core language phrases.

There are three important aspects to the interpretation of ModML programs. First, structures are flattened during translation; as we have seen in the previous chapter, the IntML language does not have any constructs for collecting declarations in any structure-like way. The interpretation of ModML programs translates declarations in ModML structures into top-level IntML declarations. To avoid name clashes, the interpretation chooses fresh variables for all generated IntML bindings and maintains a mapping from ModML identifiers to IntML variables.

The second important aspect to the interpretation of ModML programs is that functors are specialised for each application. Although there is a potential possibility for an increase in code size, ModML functors are not allowed to be recursive, thus, the interpretation terminates. Moreover, experience with large software projects that use Standard ML Modules extensively, such as the ML Kit and the Standard ML of New Jersey compiler, indicates that few functors are applied twice or more.

The third important aspect to the interpretation of ModML programs is that signature constraints are dealt with by translation environment processing, only; no IntML phrases are generated for signature constraints. In the case a value component of a structure is made less polymorphic by a signa-

ture constraint, the instantiation is captured in the translation environment and code generation for the instantiation is postponed till the value component of the constrained structure is accessed. As an example, consider the ModML program

```
datatype s = A
structure S = struct val id = fn b => b
              end : sig val id : s -> s end
val a = S.id A
```

This program translates into the IntML declaration

$$\begin{array}{l}\texttt{datdec}\ t = c\\ \texttt{valdec}\ x : \forall\alpha.\alpha \to \alpha\ \texttt{=}\ \lambda y : \alpha.y_\alpha\\ \texttt{valdec}\ z : t\ \texttt{=}\ x_{t\to t}\ c\end{array}$$

where c is an IntML constructor associated with the identifier `A`, t is a type name with arity 0, and x, y, and z are IntML variables associated with the identifiers `id`, `b`, and `a`, respectively.

In the following sections, we formalise the interpretation of ModML programs. We then proceed to demonstrate that all typable ModML programs that have no opaque signature constraints are translatable. In Section 8.7, we show type correctness of the translation. Finally, in Section 8.8, we demonstrate type soundness for ModML in the sense that if a well-typed ModML program translates into an IntML declaration then, according to the dynamic semantics of IntML, this declaration does not go wrong.

8.1 Semantic Objects

The semantic objects of the translation include the semantic objects of elaboration as defined in Figure 2.2 on page 12. The additional semantic objects are shown in Figure 8.1.

Translation value environments map value identifiers to entries of the form $(\sigma, is, a : \sigma')$, where σ is the type scheme for the value identifier, is is the identifier status of the value identifier (`v` or `c`), and $a : \sigma'$ is a pair of an IntML variable and a type scheme σ'. The type scheme σ' is the type scheme for the IntML variable a. Because a value component of a structure can be made less polymorphic by a signature constraint, the type scheme σ' can be more general than the type scheme σ.

$$
\begin{array}{rcl}
\mathcal{SE} & \in & \text{TStrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{TEnv} \\
\mathcal{VE} & \in & \text{TValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{TValEntry} \\
(\sigma, is, a : \sigma') & \in & \text{TValEntry} = \text{TypeScheme} \times \text{IdStatus} \\
 & & \quad \times (\text{IVar} \cup \text{ICon}) \times \text{TypeScheme} \\
\mathcal{E} & \in & \text{TEnv} = \text{TStrEnv} \times \text{TyEnv} \times \text{TValEnv} \\
\mathcal{F} & \in & \text{TFunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunClos} \\
fc & \in & \text{FunClos} = \text{TBasis} \times \text{StrId} \times \text{StrExp} \times \text{FunSig} \\
\mathcal{B} & \in & \text{TBasis} = \text{TFunEnv} \times \text{TEnv}
\end{array}
$$

Figure 8.1: Additional semantic objects of translation.

To postpone the interpretation of a functor body till the functor is applied, a functor identifier is mapped to a *functor closure*, which is a quadruple of the form $(\mathcal{B}, strid, strexp, \Phi)$, where $\mathcal{B}$ is a translation basis for capturing free variables of the functor, *strid* is a structure identifier for the formal parameter, *strexp* is the functor body, and Φ is the functor signature for the functor.

8.2 Weakening

When $\mathcal{A}$ is some translation environment or translation basis, we define *weakening* of $\mathcal{A}$, written $\overline{\mathcal{A}}$, to be the elaboration object derived from the translation object $\mathcal{A}$ by erasing all translation information that is not present in the corresponding elaboration object:

$$
\begin{array}{rcl}
\overline{(\mathcal{F}, \mathcal{E})} & = & (\overline{\mathcal{F}}, \overline{\mathcal{E}}) \\
\overline{\mathcal{F}} & = & \{funid \mapsto \overline{\mathcal{F}(funid)} \mid funid \in \text{Dom } \mathcal{F}\} \\
\overline{(\mathcal{B}, strid, strexp, \Phi)} & = & \Phi \\
\overline{(\mathcal{SE}, TE, \mathcal{VE})} & = & (\overline{\mathcal{SE}}, TE, \overline{\mathcal{VE}}) \\
\overline{\mathcal{SE}} & = & \{strid \mapsto \overline{\mathcal{SE}(strid)} \mid strid \in \text{Dom } \mathcal{SE}\} \\
\overline{\mathcal{VE}} & = & \{vid \mapsto \overline{\mathcal{VE}(vid)} \mid vid \in \text{Dom } \mathcal{VE}\}
\end{array}
$$

$$\overline{(\sigma, is, a : \sigma')} \;=\; (\sigma, is)$$

8.3 Enlargement

Enlargement relates translation environments much as enrichment relates elaboration environments (see Section 2.16 on page 24).

A translation value environment entry $(\sigma_1, is_1, a_1 : \sigma_1')$ *enlarges* another such object $(\sigma_2, is_2, a_2 : \sigma_2')$, written $(\sigma_1, is_1, a_1 : \sigma_1') \gg (\sigma_2, is_2, a_2 : \sigma_2')$, if $(\sigma_1, is_1) \succ (\sigma_2, is_2)$ and $(a_1 : \sigma_1') = (a_2 : \sigma_2')$.

Enlargement is extended to environments, inductively, as follows. A translation environment $\mathcal{E}_1 = (\mathcal{SE}_1, TE_1, \mathcal{VE}_1)$ *enlarges* another translation environment $\mathcal{E}_2 = (\mathcal{SE}_2, TE_2, \mathcal{VE}_2)$, written $\mathcal{E}_1 \gg \mathcal{E}_2$, if

1. Dom $\mathcal{SE}_1 \supseteq$ Dom $\mathcal{SE}_2$ and $\mathcal{SE}_1(strid) \gg \mathcal{SE}_2(strid)$ for all $strid \in$ Dom $\mathcal{SE}_2$
2. $TE_1 \succ TE_2$
3. Dom $\mathcal{VE}_1 \supseteq$ Dom $\mathcal{VE}_2$ and $\mathcal{VE}_1(vid) \gg \mathcal{VE}_2(vid)$ for all $vid \in$ Dom $\mathcal{VE}_2$

The following proposition states parts of the relationship between the notion of enrichment and the notion of enlargement:

Proposition 8.3.1 *If* $\mathcal{E} \gg \mathcal{E}'$ *then* $\overline{\mathcal{E}} \succ \overline{\mathcal{E}'}$.

PROOF The proof is a simple inductive argument over the structure of $\mathcal{E}'$. □

8.4 From ModML Core to IntML

We first present rules for translating ModML Core phrases into IntML phrases. The rules allow inferences among sentences of the form

$$\mathcal{E} \vdash exp \Rightarrow \tau, e$$

where $\mathcal{E}$ is a translation environment, *exp* is a ModML expression, τ is a type, and e is an IntML expression, and of the form

$$\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d)$$

where $\mathcal{E}$ and $\mathcal{E}'$ are translation environments, *dec* is a ModML declaration, N is a set of names, and d is an IntML declaration. Sentences of the former form are read "*exp* translates to (τ, e) in $\mathcal{E}$." Sentences of the latter form are read "*dec* translates to $(N)(\mathcal{E}', d)$ in $\mathcal{E}$." The prefix (N) in objects of the form $(N)(\mathcal{E}, d)$ binds names and we identify such objects up to renaming of bound names and deletion of names from the prefix that do not occur in the body $(\mathcal{E}, d)$.

Expressions $\boxed{\mathcal{E} \vdash exp \Rightarrow \tau, e}$

$$\frac{\mathcal{E}(longvid) = (\sigma, is, a : \sigma') \quad \sigma' \succ \tau}{\mathcal{E} \vdash longvid \Rightarrow \tau, a_\tau} \tag{8.1}$$

$$\frac{\begin{array}{c} x \notin \text{names } \mathcal{E} \quad vid \notin \text{Dom } \mathcal{E} \text{ or } is \text{ of } \mathcal{E}(vid) = \mathtt{v} \\ \mathcal{E} + \{vid \mapsto (\tau, \mathtt{v}, x : \tau)\} \vdash exp \Rightarrow \tau', e \end{array}}{\mathcal{E} \vdash \mathtt{fn^v}\ vid\ \mathtt{=>}\ exp \Rightarrow \tau \to \tau', \lambda x : \tau.e} \tag{8.2}$$

$$\frac{\mathcal{E}(longvid) = (\sigma, \mathtt{c}, c : \sigma) \quad \sigma \succ \tau \quad \mathcal{E} \vdash exp \Rightarrow \tau', e}{\mathcal{E} \vdash \mathtt{fn^c}\ longvid\ \mathtt{=>}\ exp \Rightarrow \tau \to \tau', \lambda c : \tau.e} \tag{8.3}$$

$$\frac{\mathcal{E} \vdash exp_1 \Rightarrow \tau' \to \tau, e_1 \quad \mathcal{E} \vdash exp \Rightarrow \tau', e_2}{\mathcal{E} \vdash exp_1\ exp_2 \Rightarrow \tau, e_1\ e_2} \tag{8.4}$$

$$\frac{\begin{array}{c} \mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d) \quad N \cap \text{names}(\mathcal{E}, \tau) = \emptyset \\ \mathcal{E} + \mathcal{E}' \vdash exp \Rightarrow \tau, e \end{array}}{\mathcal{E} \vdash \mathtt{let}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \Rightarrow \tau, \mathtt{let}\ d\ \mathtt{in}\ e} \tag{8.5}$$

Comment:

(8.1) The type scheme σ' is the type scheme for the IntML variable a. Because a value component of a structure can be made less polymorphic by a signature constraint, the type scheme σ' can be more general than the type scheme σ, which is the type scheme for *longvid*. A global well-formedness condition on $\mathcal{E}$ ensures that we also have $\sigma \succ \tau$. We shall later see that this well-formedness condition is maintained by the rules.

Declarations $\boxed{\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d)}$

$$\frac{\begin{array}{c}\text{tyvars } \alpha^{(k)} \cap \text{tyvars } \mathcal{E} = \emptyset \quad \sigma = \forall\alpha^{(k)}.\tau \\ \mathcal{E} \vdash exp \Rightarrow \tau, e \quad \mathcal{E}' = \{vid \mapsto (\sigma, \mathtt{v}, x : \sigma)\} \quad x \notin \text{names } \mathcal{E}\end{array}}{\mathcal{E} \vdash \mathtt{val}\ vid\ \mathtt{=}\ exp \Rightarrow (\{x\})(\mathcal{E}', \mathtt{valdec}\ x : \alpha^{(k)}.\tau\ \mathtt{=}\ e)} \tag{8.6}$$

$$\frac{\begin{array}{c}tyvarseq = \alpha^{(k)} \quad \text{arity } t = k \quad \sigma = \forall\alpha^{(k)}.\alpha^{(k)}t \\ VE = \{vid \mapsto (\sigma, \mathtt{c})\} \quad \mathcal{VE} = \{vid \mapsto (\sigma, \mathtt{c}, c : \sigma)\} \\ \mathcal{E}' = (\{tycon \mapsto (t, VE)\}, \mathcal{VE})\end{array}}{\mathcal{E} \vdash \mathtt{datatype}\ tyvarseq\ tycon\ \mathtt{=}\ vid \Rightarrow (\{t, c\})(\mathcal{E}', \mathtt{datdec}\ \alpha^{(k)}t\ \mathtt{=}\ c)} \tag{8.7}$$

$$\frac{tyvarseq = \alpha^{(k)} \quad \overline{\mathcal{E}} \vdash ty \Rightarrow \tau \quad \mathcal{E}' = \{tycon \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}}{\mathcal{E} \vdash \mathtt{type}\ tyvarseq\ tycon\ \mathtt{=}\ ty \Rightarrow (\emptyset)(\mathcal{E}', \varepsilon)} \tag{8.8}$$

$$\frac{\mathcal{E}(longstrid) = \mathcal{E}' \quad \text{Dom } \mathcal{E}' = I}{\mathcal{E} \vdash \mathtt{open}^I\ longstrid \Rightarrow (\emptyset)(\mathcal{E}', \varepsilon)} \tag{8.9}$$

Comment:

(8.8) and (8.9) No code is generated for `type` declarations or for `open` declarations.

8.5 Static Interpretation

The rules for interpreting ModML Modules phrases into IntML declarations allow inferences among sentences of the form

$$\mathcal{B} \vdash phrase \Rightarrow (N)(\mathcal{A}, d)$$

where $\mathcal{B}$ is a translation basis, *phrase* is either a structure-level declaration, a structure-level expression, or a top-level declaration, $\mathcal{A}$ is either a translation environment or a translation basis, N is a set of names, and d is an IntML declaration; sentences of this form are read "*phrase* translates to $(N)(\mathcal{A}, d)$ in $\mathcal{B}$."

Structure-level Declarations $\boxed{\mathcal{B} \vdash \mathit{strdec} \Rightarrow (N)(\mathcal{E}, d)}$

$$\frac{\mathcal{E} \vdash \mathit{dec} \Rightarrow (N)(\mathcal{E}, d)}{(\mathcal{F}, \mathcal{E}) \vdash \mathit{dec} \Rightarrow (N)(\mathcal{E}, d)} \tag{8.10}$$

$$\frac{\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N)(\mathcal{E}, d)}{\mathcal{B} \vdash \texttt{structure}\ \mathit{strid}\ \texttt{=}\ \mathit{strexp} \Rightarrow (N)(\{\mathit{strid} \mapsto \mathcal{E}\}, d)} \tag{8.11}$$

$$\frac{\begin{array}{c}\mathcal{B} \vdash \mathit{strdec}_1 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \quad (N_1 \cup N_2) \cap \mathrm{names}\ \mathcal{B} = \emptyset \\ \mathcal{B} + \mathcal{E}_1 \vdash \mathit{strdec}_2 \Rightarrow (N_2)(\mathcal{E}_2, d_2) \quad N_2 \cap (N_1 \cup \mathrm{names}(\mathcal{E}_1, d_1)) = \emptyset\end{array}}{\mathcal{B} \vdash \mathit{strdec}_1\ \mathit{strdec}_2 \Rightarrow (N_1 \cup N_2)(\mathcal{E}_1 + \mathcal{E}_2, d_1\ ;\ d_2)} \tag{8.12}$$

$$\frac{}{\mathcal{B} \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \varepsilon)} \tag{8.13}$$

Structure-level Expressions $\boxed{\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N)(\mathcal{E}, d)}$

$$\frac{\mathcal{B} \vdash \mathit{strdec} \Rightarrow (N)(\mathcal{E}, d)}{\mathcal{B} \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow (N)(\mathcal{E}, d)} \tag{8.14}$$

$$\frac{\mathcal{B}(\mathit{longstrid}) = \mathcal{E}}{\mathcal{B} \vdash \mathit{longstrid} \Rightarrow (\emptyset)(\mathcal{E}, \varepsilon)} \tag{8.15}$$

$$\frac{\begin{array}{c}\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N)(\mathcal{E}, d) \quad \overline{\mathcal{B}} \vdash \mathit{sigexp} \Rightarrow \Sigma \\ \Sigma \geq \overline{\mathcal{E}'} \quad \mathcal{E} \gg \mathcal{E}' \quad N \cap \mathrm{names}\ \mathcal{B} = \emptyset\end{array}}{\mathcal{B} \vdash \mathit{strexp}\ :\ \mathit{sigexp} \Rightarrow (N)(\mathcal{E}', d)} \tag{8.16}$$

$$\frac{\begin{array}{c}\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N)(\mathcal{E}, d) \quad \mathcal{B}(\mathit{funid}) = (\mathcal{B}_0, \mathit{strid}, \mathit{strexp}_0, \Phi) \\ \Phi \geq (\overline{\mathcal{E}'}, (T')\overline{\mathcal{E}_1}) \quad T' \subseteq N_1 \quad \mathcal{E} \gg \mathcal{E}' \\ (N \cup N_1) \cap \mathrm{names}\ \mathcal{B} = \emptyset \quad N_1 \cap (N \cup \mathrm{names}(\mathcal{E}, d)) = \emptyset \\ \mathcal{B}_0 + \{\mathit{strid} \mapsto \mathcal{E}'\} \vdash \mathit{strexp}_0 \Rightarrow (N_1)(\mathcal{E}_1, d_1)\end{array}}{\mathcal{B} \vdash \mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)} \Rightarrow (N \cup N_1)(\mathcal{E}_1, d\ ;\ d_1)} \tag{8.17}$$

Top-level Declarations $\boxed{\mathcal{B} \vdash topdec \Rightarrow (N)(\mathcal{B}', d)}$

$$\frac{\mathcal{B} \vdash strdec \Rightarrow (N)(\mathcal{E}, d)}{\mathcal{B} \vdash strdec \Rightarrow (N)((\{\}, \mathcal{E}), d)} \tag{8.18}$$

$$\frac{\begin{array}{c}\overline{\mathcal{B}} \vdash sigexp \Rightarrow (T)E \quad T \cap \text{tynames } \mathcal{B} = \emptyset \\ \overline{\mathcal{B}} + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma \\ \mathcal{F} = \{funid \mapsto (\mathcal{B}, strid, strexp, (T)(E, \Sigma))\}\end{array}}{\mathcal{B} \vdash \texttt{functor}\ funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ strexp \Rightarrow (\emptyset)((\mathcal{F}, \{\}), \varepsilon)} \tag{8.19}$$

$$\frac{\begin{array}{c}\mathcal{B} \vdash topdec_1 \Rightarrow (N_1)(\mathcal{B}_1, d_1) \quad (N_1 \cup N_2) \cap \text{names } \mathcal{B} = \emptyset \\ \mathcal{B} + \mathcal{B}_1 \vdash topdec_2 \Rightarrow (N_2)(\mathcal{B}_2, d_2) \quad N_2 \cap (N_1 \cup \text{names}(\mathcal{B}_1, d_1)) = \emptyset\end{array}}{\mathcal{B} \vdash topdec_1\ topdec_2 \Rightarrow (N_1 \cup N_2)(\mathcal{B}_1 + \mathcal{B}_2, d_1\ ;\ d_2)} \tag{8.20}$$

$$\frac{}{\mathcal{B} \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \varepsilon)} \tag{8.21}$$

Comment:

(8.19) Interpretation of the functor body is delayed until the functor is applied.

8.6 Translatability

In this section, we demonstrate that if a ModML program is typable under some assumptions B, according to the static semantics for ModML presented in Chapter 2, then the ModML program may be translated into an IntML declaration, under assumptions that are related to B. To be able to state a proposition expressing this property, we first need to define a notion of well-formedness of translation environments and translation bases.

Well-formedness of translation environments and translation bases expresses that (1) the type scheme for a value identifier is an instance of the type scheme for the associated IntML variable (or IntML constructor), (2) if the identifier status for a value identifier denotes a constructor then the value

identifier is associated with an IntML constructor, and finally, (3) a functor body in a functor closure must be translatable under appropriate assumptions with appropriate result. Well-formedness of a translation environment or a translation basis $\mathcal{A}$ is written $\vdash \mathcal{A}$ and is defined inductively by the following inference rules:

Well-Formedness $\boxed{\vdash \mathcal{A}}$

$$\frac{\vdash \mathcal{E} \quad \vdash \mathcal{F}(\mathit{funid}) \text{ for all } \mathit{funid} \in \mathrm{Dom}\ \mathcal{F}}{\vdash (\mathcal{F}, \mathcal{E})} \tag{8.22}$$

$$\frac{\vdash \mathcal{SE}(\mathit{strid}) \text{ for all } \mathit{strid} \in \mathrm{Dom}\ \mathcal{SE} \quad \vdash \mathcal{VE}(\mathit{vid}) \text{ for all } \mathit{vid} \in \mathrm{Dom}\ \mathcal{VE}}{\vdash (\mathcal{SE}, TE, \mathcal{VE})} \tag{8.23}$$

$$\frac{\sigma' \succ \sigma}{\vdash (\sigma, \mathtt{v}, a : \sigma')} \tag{8.24}$$

$$\frac{a \in \mathrm{ICon}}{\vdash (\sigma, \mathtt{c}, a : \sigma)} \tag{8.25}$$

$$\frac{\begin{array}{c}\vdash \mathcal{B} \\ \forall (E, (T)E') \leq \Phi,\ \forall \mathcal{E}.\ (\ \vdash \mathcal{E}\ \wedge \overline{\mathcal{E}} = E\ )\ .\ \exists (N)(\mathcal{E}', d)\ . \\ (\ \mathcal{B} + \{\mathit{strid} \mapsto \mathcal{E}\} \vdash \mathit{strexp} \Rightarrow (N)(\mathcal{E}', d)\ \wedge N \supseteq T\ \wedge \\ \vdash \mathcal{E}'\ \wedge\ \overline{\mathcal{E}'} = E'\ )\end{array}}{\vdash (\mathcal{B}, \mathit{strid}, \mathit{strexp}, \Phi)} \tag{8.26}$$

As discussed in Section 8.3, there is a close relationship between the notion of enrichment for the static semantics of ModML and the notion of enlargement; indeed, the following proposition holds:

Proposition 8.6.1 *If* $\vdash \mathcal{E}$ *and* $\overline{\mathcal{E}} \succ E$ *then there exists* $\mathcal{E}'$ *such that* $\mathcal{E} \gg \mathcal{E}'$ *and* $\overline{\mathcal{E}'} = E$ *and* $\vdash \mathcal{E}'$.

PROOF The proof is a simple inductive argument over the structure of E, using transitivity of type scheme generalisation. □

The following proposition states that if a ModML declaration is typable under some assumptions E then the declaration is also translatable under assumptions related to E.

Proposition 8.6.2 (Core translatability) *If* $\overline{\mathcal{E}} \vdash dec \Rightarrow (T)E'$ *and* $\vdash \mathcal{E}$ *then there exists* $(N)(\mathcal{E}', d)$ *such that* $\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d)$ *and* $N \supseteq T$ *and* $\vdash \mathcal{E}'$ *and* $\overline{\mathcal{E}'} = E'$. *Moreover, if* $\overline{\mathcal{E}} \vdash exp \Rightarrow \tau$ *and* $\vdash \mathcal{E}$ *then there exists* e *such that* $\mathcal{E} \vdash exp \Rightarrow \tau, e$.

PROOF The proof is by induction on the structure of *dec* and *exp*.

CASE $exp = longvid$ From assumptions and from rule 2.4, we have

$$\overline{\mathcal{E}}(longvid) = (\sigma, is) \tag{8.27}$$

$$\sigma \succ \tau \tag{8.28}$$

$$\vdash \mathcal{E} \tag{8.29}$$

From (8.27) and from the definition of weakening we have that there exist σ' and a such that

$$\mathcal{E}(longvid) = (\sigma, is, a : \sigma') \tag{8.30}$$

Moreover, from (8.29), from (8.30), from the definition of well-formedness, and from reflexivity of type scheme generalisation, we have $\sigma' \succ \sigma$. Thus, from (8.28) and from transitivity of generalisation, we have

$$\sigma' \succ \tau \tag{8.31}$$

Now, from rule 8.1 and from (8.30) and (8.31), we have $\mathcal{E} \vdash exp \Rightarrow \tau, a_\tau$, as required.

CASE $exp = \texttt{fn}^{\texttt{c}}\ vid\ \texttt{=>}\ exp'$ From assumptions and from rule 2.6, we have

$$\overline{\mathcal{E}}(longvid) = (\sigma, \texttt{c}) \tag{8.32}$$

$$\sigma \succ \tau_1 \tag{8.33}$$

$$\overline{\mathcal{E}} \vdash exp' \Rightarrow \tau' \tag{8.34}$$

$$\overline{\mathcal{E}} \vdash exp \Rightarrow \tau_1 \to \tau'$$

$$\vdash \mathcal{E} \tag{8.35}$$

From (8.32), from (8.35), from the definition of well-formedness, and from the definition of weakening, we have there exists $c \in \mathrm{ICon}$ such that

$$\mathcal{E}(longvid) = (\sigma, \mathtt{c}, c : \sigma) \tag{8.36}$$

By applying induction to (8.34) and (8.35), we have there exists e such that

$$\mathcal{E} \vdash exp' \Rightarrow \tau', e \tag{8.37}$$

Now, from rule 8.3 and from (8.36), (8.33), and (8.37), we have $\mathcal{E} \vdash exp \Rightarrow \tau_1 \rightarrow \tau', \lambda c : \tau_1.e$, as required.

CASE $exp = \mathtt{let}\ dec\ \mathtt{in}\ exp'\ \mathtt{end}$ From assumptions and from rule 2.8, we have

$$\overline{\mathcal{E}} \vdash dec \Rightarrow (T)E' \tag{8.38}$$
$$\overline{\mathcal{E}} + E' \vdash exp' \Rightarrow \tau \tag{8.39}$$
$$T \cap \mathrm{tynames}(\overline{\mathcal{E}}, \tau) = \emptyset \tag{8.40}$$
$$\vdash \mathcal{E} \tag{8.41}$$

By applying induction to (8.38) and (8.41), we have there exists $(N)(\mathcal{E}', d)$ such that

$$\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d) \tag{8.42}$$
$$N \supseteq T \tag{8.43}$$
$$\vdash \mathcal{E}' \tag{8.44}$$
$$\overline{\mathcal{E}'} = E' \tag{8.45}$$

By appropriate renaming of bound names, we have

$$N \cap \mathrm{names}(\mathcal{E}, \tau) = \emptyset \tag{8.46}$$

From (8.41), from (8.44), and from the definition of well-formedness, we have

$$\vdash (\mathcal{E} + \mathcal{E}') \tag{8.47}$$

Now, from (8.39), from (8.45), and from the definition of weakening, we have

$$\overline{\mathcal{E} + \mathcal{E}'} \vdash exp' \Rightarrow \tau \tag{8.48}$$

By applying induction to (8.47) and (8.48), we have there exists e such that

$$\mathcal{E} + \mathcal{E}' \vdash exp' \Rightarrow \tau, e \tag{8.49}$$

Now, from rule 8.5 and from (8.42), (8.46), and (8.49), we have $\mathcal{E} \vdash exp \Rightarrow \tau, \texttt{let}\ d\ \texttt{in}\ e$, as required.

CASE $dec = \texttt{val}\ vid\ \texttt{=}\ exp$ From assumptions and from rule 2.9, we have

$$\overline{\mathcal{E}} \vdash exp \Rightarrow \tau \tag{8.50}$$
$$\text{tyvars}\ \alpha^{(k)} \cap \text{tyvars}\ \overline{\mathcal{E}} = \emptyset \tag{8.51}$$
$$\overline{\mathcal{E}} \vdash dec \Rightarrow (\emptyset)E'$$
$$\sigma = \forall\alpha^{(k)}.\tau \tag{8.52}$$
$$E' = \{vid \mapsto (\sigma, \texttt{v})\} \tag{8.53}$$
$$\vdash \mathcal{E} \tag{8.54}$$

By appropriate renaming of bound type variables, we have

$$\text{tyvars}\ \alpha^{(k)} \cap \text{tyvars}\ \mathcal{E} = \emptyset \tag{8.55}$$

By applying induction to (8.50) and (8.54), we have there exists e such that

$$\mathcal{E} \vdash exp \Rightarrow \tau, e \tag{8.56}$$

Now, choose $x \in \text{ICon}$ such that

$$x \notin \text{names}\ \mathcal{E} \tag{8.57}$$

Let $\mathcal{E}' = \{vid \mapsto (\sigma, \texttt{v}, x : \sigma)\}$. From reflexivity of type scheme generalisation and from the definition of well-formedness, we have $\vdash \mathcal{E}'$. Moreover, from the definition of weakening and from (8.53), we have $\overline{\mathcal{E}'} = E'$. It follows from rule 8.6 and from (8.55), (8.52), (8.56), and (8.57) that we have $\mathcal{E} \vdash dec \Rightarrow (\{x\})(\mathcal{E}', \texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e)$, as required.

CASE $dec = \texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid$ From assumptions and from rule 2.10, we have

$$tyvarseq = \alpha^{(k)} \tag{8.58}$$
$$\text{arity}\ t = k \tag{8.59}$$
$$VE = \{vid \mapsto (\sigma, \texttt{c})\} \tag{8.60}$$
$$E' = (\{tycon \mapsto (t, VE)\}, VE) \tag{8.61}$$
$$\sigma = \forall\alpha^{(k)}.\alpha^{(k)}t \tag{8.62}$$
$$T = \{t\} \tag{8.63}$$

Let $\mathcal{VE} = \{vid \mapsto (\sigma, \mathtt{c}, c : \sigma)\}$ and let $\mathcal{E}' = (\{tycon \mapsto (t, VE), \mathcal{VE})$. From rule 8.7 and from (8.58), (8.59), (8.62), and (8.60), we have

$$\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', \mathtt{datdec}\ \alpha^{(k)}t\ \mathtt{=}\ c) \tag{8.64}$$

$$N = \{t, c\} \tag{8.65}$$

From (8.63) and from (8.65), we have $N \supseteq T$, as required. Moreover, from the definition of well-formedness, we have $\vdash \mathcal{E}'$, as required. Further, from the definition of weakening and from (8.60) and (8.61), we have $\overline{\mathcal{E}'} = E'$, as required.

The proofs for the remaining cases follow similarly. □

The previous proposition extends to other ModML phrases. But before we state such a proposition for ModML, we shall first demonstrate the following proposition concerning realisation of functor bodies:

Proposition 8.6.3 (Functor instance typeability) *If* $B + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma$ *and* $T \cap \mathrm{tynames}\ B = \emptyset$ *and* $(T)(E, \Sigma) \geq (E', \Sigma')$ *then* $B + \{strid \mapsto E'\} \vdash strexp \Rightarrow \Sigma'$.

Proof From the definition of functor signature instantiation in Section 2.15 on page 23 and from assumptions, we have that there exists a realisation φ such that

$$\varphi(E, \Sigma) = (E', \Sigma') \tag{8.66}$$

$$\mathrm{Supp}\ \varphi \subseteq T \tag{8.67}$$

From Proposition 3.1.11 and from assumptions, we have

$$\varphi(B + \{strid \mapsto E\}) \vdash strexp \Rightarrow \varphi(\Sigma) \tag{8.68}$$

Now, from assumptions, we have $T \cap \mathrm{tynames}\ B = \emptyset$, thus, from (8.66), (8.67), and (8.68), we have $B + \{strid \mapsto E'\} \vdash strexp \Rightarrow \Sigma'$, as required. □

Using the previous proposition we can now demonstrate that if a ModML phrase is typable under some assumptions B then the phrase is translatable under assumptions related to B.

Proposition 8.6.4 (Module translatability) *Let phrase be either a structure-level declaration, a structure-level expression, or a top-level declaration. Moreover, let A be either an elaboration environment or an elaboration basis and let $\mathcal{A}$ be either a translation environment or a translation basis. If $\overline{\mathcal{B}} \vdash phrase \Rightarrow (T)A$ and $\vdash \mathcal{B}$ then there exists $(N)(\mathcal{A}, d)$ such that $\mathcal{B} \vdash phrase \Rightarrow (N)(\mathcal{A}, d)$ and $N \supseteq T$ and $\vdash \mathcal{A}$ and $\overline{\mathcal{A}} = A$.*

PROOF The proof is by induction over the structure of *strdec*, *strexp*, and *topdec*.

CASE $strdec = dec$ The desired result follows directly from rule 2.28, rule 8.10, and Proposition 8.6.2.

CASE $strdec = strdec_1\ strdec_2$ From assumptions and from rule 2.30, we have

$$\overline{\mathcal{B}} \vdash strdec_1 \Rightarrow (T_1)E_1 \tag{8.69}$$
$$(T_1 \cup T_2) \cap \text{tynames } \overline{\mathcal{B}} = \emptyset \tag{8.70}$$
$$\overline{\mathcal{B}} + E_1 \vdash strdec_2 \Rightarrow (T_2)E_2 \tag{8.71}$$
$$T_2 \cap (T_1 \cup \text{tynames } E_1) = \emptyset \tag{8.72}$$
$$\overline{\mathcal{B}} \vdash strdec \Rightarrow (T_1 \cup T_2)(E_1 + E_2)$$
$$\vdash \mathcal{B} \tag{8.73}$$

We can apply induction to (8.69) and (8.73) to get there exists $(N_1)(\mathcal{E}_1, d_1)$ such that

$$\mathcal{B} \vdash strdec_1 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \tag{8.74}$$
$$N_1 \supseteq T_1 \tag{8.75}$$
$$\vdash \mathcal{E}_1 \tag{8.76}$$
$$\overline{\mathcal{E}_1} = E_1 \tag{8.77}$$

By appropriate renaming of bound names, we can assume

$$N_1 \cap \text{names } \mathcal{B} = \emptyset \tag{8.78}$$

From the definition of weakening and from (8.71) and (8.77), we have

$$\overline{\mathcal{B} + \mathcal{E}_1} \vdash strdec_2 \Rightarrow (T_2)E_2 \tag{8.79}$$

Moreover, from (8.76) and (8.73) and from the definition of well-formedness, we have

$$\vdash (\mathcal{B} + \mathcal{E}_1) \tag{8.80}$$

We can now apply induction to (8.79) and (8.80) to get there exists $(N_2)(\mathcal{E}_2, d_2)$ such that

$$\mathcal{B} + \mathcal{E}_1 \vdash \mathit{strdec}_2 \Rightarrow (N_2)(\mathcal{E}_2, d_2) \tag{8.81}$$
$$N_2 \supseteq T_2 \tag{8.82}$$
$$\vdash \mathcal{E}_2 \tag{8.83}$$
$$\overline{\mathcal{E}_2} = E_2 \tag{8.84}$$

By appropriate renaming of bound names, we can assume

$$N_2 \cap \text{names } \mathcal{B} = \emptyset \tag{8.85}$$
$$N_2 \cap (N_1 \cup \text{names}(\mathcal{E}_1, d_1)) = \emptyset \tag{8.86}$$

Now, from rule 8.12 and from (8.74), (8.78), (8.85), (8.81), and (8.86), we have

$$\mathcal{B} \vdash \mathit{strdec} \Rightarrow (N_1 \cup N_2)(\mathcal{E}_1 + \mathcal{E}_2, d_1 \ ; \ d_2)$$

Moreover, from (8.75) and (8.82), we have $(N_1 \cup N_2) \supseteq (T_1 \cup T_2)$ and from (8.76), from (8.83), and from the definition of well-formedness, we have $\vdash (\mathcal{E}_1 + \mathcal{E}_2)$. From (8.77) and from (8.84), we also have $\overline{\mathcal{E}_1 + \mathcal{E}_2} = E_1 + E_2$, as required.

CASE $\mathit{strexp} = \mathit{strexp}' : \mathit{sigexp}$ From assumptions and from rule 2.25, we have

$$\overline{\mathcal{B}} \vdash \mathit{strexp}' \Rightarrow (T)E \tag{8.87}$$
$$\overline{\mathcal{B}} \vdash \mathit{sigexp} \Rightarrow \Sigma \tag{8.88}$$
$$\Sigma \geq E' \prec E \tag{8.89}$$
$$T \cap \text{tynames } \overline{\mathcal{B}} = \emptyset \tag{8.90}$$
$$\overline{\mathcal{B}} \vdash \mathit{strexp} \Rightarrow (T)E'$$
$$\vdash \mathcal{B} \tag{8.91}$$

By applying induction to (8.91) and (8.87), we have there exists $(N)(\mathcal{E}, d)$ such that

$$\mathcal{B} \vdash \mathit{strexp}' \Rightarrow (N)(\mathcal{E}, d) \tag{8.92}$$

$$N \supseteq T \tag{8.93}$$
$$\vdash \mathcal{E} \tag{8.94}$$
$$\overline{\mathcal{E}} = E \tag{8.95}$$

By appropriate renaming of bound names, we have

$$N \cap \text{names } \mathcal{B} = \emptyset \tag{8.96}$$

From Proposition 8.6.1 and from (8.95), (8.89), and (8.94), we have there exists $\mathcal{E}'$ such that

$$\mathcal{E} \gg \mathcal{E}' \quad \overline{\mathcal{E}'} = E' \tag{8.97}$$
$$\vdash \mathcal{E}' \tag{8.98}$$

Now, from rule 8.16 and from (8.88), (8.92), (8.89), (8.97), and (8.96), we have $\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N)(\mathcal{E}', d)$, as required.

CASE $\mathit{strexp} = \mathit{longstrid}$ From assumptions and from rule 2.24, we have

$$\overline{\mathcal{B}}(\mathit{longstrid}) = E \tag{8.99}$$
$$\overline{\mathcal{B}} \vdash \mathit{strexp} \Rightarrow (\emptyset)E$$
$$\vdash \mathcal{B} \tag{8.100}$$

From the definition of weakening and from (8.99), we have there exists $\mathcal{E}$ such that

$$\mathcal{B}(\mathit{longstrid}) = \mathcal{E} \tag{8.101}$$
$$\overline{\mathcal{E}} = E \tag{8.102}$$

Now, from rule 8.15 and from (8.101), we have

$$\mathcal{B} \vdash \mathit{strexp} \Rightarrow (\emptyset)(\mathcal{E}, \varepsilon)$$

Moreover, from (8.100) and from (8.101), we have $\vdash \mathcal{E}$, as required.

CASE $\mathit{strexp} = \mathit{funid}$ (strexp') From assumptions and from rule 2.27, we have

$$\overline{\mathcal{B}} \vdash \mathit{strexp}' \Rightarrow (T)E \tag{8.103}$$
$$\overline{\mathcal{B}}(\mathit{funid}) = \Phi \quad \Phi \geq (E'', (T')E') \tag{8.104}$$
$$E \succ E'' \tag{8.105}$$
$$(T \cup T') \cap \text{tynames } \overline{\mathcal{B}} = \emptyset \tag{8.106}$$
$$\overline{\mathcal{B}} \vdash \mathit{strexp} \Rightarrow (T \cup T')E'$$
$$\vdash \mathcal{B} \tag{8.107}$$

We can apply induction to (8.107) and (8.103) to get there exists $(N)(\mathcal{E}, d)$ such that

$$\mathcal{B} \vdash \mathit{strexp}' \Rightarrow (N)(\mathcal{E}, d) \tag{8.108}$$
$$N \supseteq T \tag{8.109}$$
$$\vdash \mathcal{E} \tag{8.110}$$
$$\overline{\mathcal{E}} = E \tag{8.111}$$

By appropriate renaming of bound names, we can assume

$$N \cap \text{names } \mathcal{B} = \emptyset \tag{8.112}$$

From (8.105) and from (8.111), we have $\overline{\mathcal{E}} \succ E''$, hence, from Proposition 8.6.1 and from (8.110), we have there exists $\mathcal{E}'$ such that

$$\mathcal{E} \gg \mathcal{E}' \quad \overline{\mathcal{E}'} = E'' \tag{8.113}$$
$$\vdash \mathcal{E}' \tag{8.114}$$

From (8.104) and from the definition of weakening, we have there exists a functor closure $cl = (\mathcal{B}_0, \mathit{strid}, \mathit{strexp}_0, \Phi')$ such that $\Phi = \Phi'$ and

$$\mathcal{B}(\mathit{funid}) = cl \tag{8.115}$$

Now, from (8.107), from (8.115), from the definition of well-formedness, and from (8.104), (8.114), and (8.113), we have that there exists $(N_1)(\mathcal{E}_1, d_1)$ such that

$$\mathcal{B}_0 + \{\mathit{strid} \mapsto \mathcal{E}'\} \vdash \mathit{strexp}_0 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \tag{8.116}$$
$$N_1 \supseteq T' \tag{8.117}$$
$$\vdash \mathcal{E}_1 \tag{8.118}$$
$$\overline{\mathcal{E}_1} = E' \tag{8.119}$$

By appropriate renaming of bound names, we can assume

$$N_1 \cap \text{names } \mathcal{B} = \emptyset \tag{8.120}$$
$$N_1 \cap (N \cup \text{names}(\mathcal{E}, d)) = \emptyset \tag{8.121}$$

Now, from rule 8.17 and from (8.115), (8.104), (8.117), (8.119), (8.108), (8.113), (8.112), (8.120), (8.116), and (8.121), we have

$$\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N \cup N_1)(\mathcal{E}_1, d \ ; \ d_1)$$

Moreover, from (8.109) and from (8.117), we have $(N \cup N_1) \supseteq (T \cup T')$, as required.

CASE *topdec* = `functor` *funid* `(` *strid* `:` *sigexp* `)` `=` *strexp* From assumptions and from rule 2.33, we have

$$\overline{\mathcal{B}} \vdash \mathit{sigexp} \Rightarrow (T)E \tag{8.122}$$
$$T \cap \text{tynames}\ \overline{\mathcal{B}} = \emptyset \tag{8.123}$$
$$\overline{\mathcal{B}} + \{\mathit{strid} \mapsto E\} \vdash \mathit{strexp} \Rightarrow \Sigma \tag{8.124}$$
$$\Phi = (T)(E, \Sigma) \tag{8.125}$$
$$F = \{\mathit{funid} \mapsto \Phi\} \tag{8.126}$$
$$\overline{\mathcal{B}} \vdash \mathit{topdec} \Rightarrow (\emptyset)(F, \{\})$$
$$\vdash \mathcal{B} \tag{8.127}$$

By appropriate renaming of bound names, we can assume

$$T \cap \text{names}\ \mathcal{B} = \emptyset \tag{8.128}$$

Let $(E_1, (T_2)E_2)$ be a functor instance such that

$$\Phi \geq (E_1, (T_2)E_2) \tag{8.129}$$

Moreover, let $\mathcal{E}_1$ be a translation environment such that

$$\vdash \mathcal{E}_1 \tag{8.130}$$
$$\overline{\mathcal{E}_1} = E_1 \tag{8.131}$$

From Proposition 8.6.3 and from (8.124), (8.123), and (8.129), we have

$$\overline{\mathcal{B}} + \{\mathit{strid} \mapsto E_1\} \vdash \mathit{strexp} \Rightarrow (T_2)E_2 \tag{8.132}$$

It now follows from (8.131), from (8.132), and from the definition of weakening that we have

$$\overline{\mathcal{B} + \{\mathit{strid} \mapsto \mathcal{E}_1\}} \vdash \mathit{strexp} \Rightarrow (T_2)E_2 \tag{8.133}$$

Moreover, from (8.127), from (8.130), and from the definition of well-formedness, we have

$$\vdash (\mathcal{B} + \{\mathit{strid} \mapsto \mathcal{E}_1\}) \tag{8.134}$$

We can now apply induction to (8.133) and (8.134) to get there exists $(N_2)(\mathcal{E}_2, d_2)$ such that

$$\mathcal{B} + \{strid \mapsto \mathcal{E}_1\} \vdash strexp \Rightarrow (N_2)(\mathcal{E}_2, d_2) \tag{8.135}$$

$$N_2 \supseteq T_2 \tag{8.136}$$

$$\vdash \mathcal{E}_2 \tag{8.137}$$

$$\overline{\mathcal{E}_2} = E_2 \tag{8.138}$$

Now, let $\mathcal{F}$ be the translation functor environment

$$\mathcal{F} = \{funid \mapsto (\mathcal{B}, strid, strexp, \Phi)\} \tag{8.139}$$

It follows from (8.126), from (8.139), and from the definition of weakening that we have

$$\overline{(\mathcal{F}, \{\})} = (F, \{\}) \tag{8.140}$$

Moreover, from the definition of well-formedness and from (8.135), (8.136), (8.137), (8.138), and (8.139), we have

$$\vdash (\mathcal{F}, \{\}) \tag{8.141}$$

From rule 8.19 and from (8.122), (8.124), (8.139), (8.125), and (8.128), we have

$$\mathcal{B} \vdash topdec \Rightarrow (\emptyset)((\mathcal{F}, \{\}), \varepsilon)$$

as required.

CASE $topdec = topdec_1\ topdec_2$ The proof for this case is similar to the proof for the case where $strdec = strdec_1\ strdec_2$.

The proofs for each of the remaining cases either follow directly or follow directly by induction. □

8.7 Type Correctness

We now demonstrate that if a ModML program is translated into an IntML declaration under some assumptions $\mathcal{B}$ then the IntML declaration is typable under assumptions that are related to $\mathcal{B}$. The way in which assumptions must be related are expressed by a type consistency relation. We write the relation

$$\Gamma \vdash_{\text{tc}} \mathcal{A}$$

where $\mathcal{A}$ is either a translation environment or a translation basis and where Γ is a type environment. Type consistency expresses that all IntML variables and IntML constructors in value entries in $\mathcal{A}$ are associated to the same type schemes as in Γ. The relation is defined inductively by the following equations:

- $\Gamma \vdash_{\mathrm{tc}} (\mathcal{SE}, TE, \mathcal{VE})$ iff $\Gamma \vdash_{\mathrm{tc}} \mathcal{E}$ for all $\mathcal{E} \in \mathrm{Ran}\ \mathcal{SE}$ and $\Gamma(x) = \sigma'$ for all $(\sigma, is, x : \sigma') \in \mathrm{Ran}\ \mathcal{VE}$ and $\Gamma(t)(c) = \sigma'$ and $\sigma' = \forall\alpha^{(k)}.\alpha^{(k)}t$ for all $(\sigma, is, c : \sigma') \in \mathrm{Ran}\ \mathcal{VE}$
- $\Gamma \vdash_{\mathrm{tc}} (\mathcal{F}, \mathcal{E})$ iff $\Gamma \vdash_{\mathrm{tc}} \mathcal{E}$ and $\Gamma \vdash_{\mathrm{tc}} \mathcal{B}$ for all $(\mathcal{B}, strid, strexp, \Phi) \in \mathrm{Ran}\ \mathcal{F}$

To prove type correctness of the translation, we must first demonstrate some properties of type consistency.

Proposition 8.7.1 *If* $\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{E}$ *and* $\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset$ *then* $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E}$.

PROOF The proof is by induction over the structure of $\mathcal{E}$. Write $\mathcal{E}$ in the form $(\mathcal{SE}, TE, \mathcal{VE})$. From assumptions and from the definition of type consistency, we have

$$\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{E} \text{ for all } \mathcal{E} \in \mathrm{Ran}\ \mathcal{SE} \tag{8.142}$$

$$\Gamma_1(x) = \sigma' \text{ for all } (\sigma, is, x : \sigma') \in \mathrm{Ran}\ \mathcal{VE} \tag{8.143}$$

$$\Gamma_1(t)(c) = \sigma' \text{ and } \sigma = \forall\alpha^{(k)}.\alpha^{(k)}t \text{ for all } (\sigma, is, c : \sigma') \in \mathrm{Ran}\ \mathcal{VE} \tag{8.144}$$

$$\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset \tag{8.145}$$

By applying induction for each $\mathcal{E} \in \mathrm{Ran}\ \mathcal{SE}$, we have from (8.142) and (8.145) that

$$\Gamma_1 + \Gamma_2 \vdash_{\mathrm{tc}} \mathcal{E} \text{ for all } \mathcal{E} \in \mathrm{Ran}\ \mathcal{SE} \tag{8.146}$$

From (8.145) and from (8.143), we have

$$(\Gamma_1 + \Gamma_2)(x) = \sigma' \text{ for all } (\sigma, is, x : \sigma') \in \mathrm{Ran}\ \mathcal{VE} \tag{8.147}$$

Moreover, from (8.145) and from (8.144), we have

$$(\Gamma_1 + \Gamma_2)(t)(c) = \sigma' \text{ and } \sigma' = \forall\alpha^{(k)}.\alpha^{(k)}t \text{ for all } (\sigma, is, c : \sigma') \in \mathrm{Ran}\ \mathcal{VE} \tag{8.148}$$

Now, from the definition of type consistency and from (8.146), (8.147), and (8.148), we have $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E}$, as required. □

The preceding proposition extends to bases as follows:

Proposition 8.7.2 *If* $\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{B}$ *and* $\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset$ *then* $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B}$.

PROOF The proof is by induction over the structure of $\mathcal{B}$. Write $\mathcal{B}$ in the form $(\mathcal{F}, \mathcal{E})$. From assumptions and from the definition of type consistency, we have

$$\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{E} \tag{8.149}$$

$$\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{B} \text{ for all } (\mathcal{B}, \mathit{strid}, \mathit{strexp}, \Phi) \in \mathrm{Ran}\ \mathcal{F} \tag{8.150}$$

$$\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset \tag{8.151}$$

From Proposition 8.7.1 and from (8.149) and (8.151), we have

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E} \tag{8.152}$$

Moreover, from (8.150) and (8.151), we can apply induction for each $\mathcal{B} \in (\mathcal{B}$ of Ran $\mathcal{F})$ to get

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B} \text{ for all } (\mathcal{B}, \mathit{strid}, \mathit{strexp}, \Phi) \in \mathrm{Ran}\ \mathcal{F} \tag{8.153}$$

Now, from the definition of type consistency and from (8.152) and (8.153), we have $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B}$, as required. □

The following proposition states that type consistency is closed under extension of translation environments.

Proposition 8.7.3 *If* $\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{E}_1$ *and* $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E}_2$ *and* $\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset$ *then* $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E}_1 + \mathcal{E}_2$.

PROOF Write $\mathcal{E}_1$ in the form $(\mathcal{SE}_1, TE_1, \mathcal{VE}_1)$ and write $\mathcal{E}_2$ in the form $(\mathcal{SE}_2, TE_2, \mathcal{VE}_2)$. From assumptions and from the definition of type consistency, we have

$$\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{E} \text{ for all } \mathcal{E} \in \mathrm{Ran}\ \mathcal{SE}_1 \tag{8.154}$$

$$\Gamma_1(x) = \sigma' \text{ for all } (\sigma, \mathit{is}, x : \sigma') \in \mathrm{Ran}\ \mathcal{VE}_1 \tag{8.155}$$

$$\Gamma_1(t)(c) = \sigma' \text{ and } \sigma' = \forall \alpha^{(k)}.\alpha^{(k)}t \text{ for all } (\sigma, \mathit{is}, a : \sigma') \in \mathrm{Ran}\ \mathcal{VE}_1 \tag{8.156}$$

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E} \text{ for all } \mathcal{E} \in \mathrm{Ran}\ \mathcal{SE}_2 \tag{8.157}$$

$$(\Gamma_1 + \Gamma_2)(x) = \sigma' \text{ for all } (\sigma, \mathit{is}, x : \sigma') \in \mathrm{Ran}\ \mathcal{VE}_2 \tag{8.158}$$

$$(\Gamma_1 + \Gamma_2)(t)(c) = \sigma' \text{ and } \sigma' = \forall \alpha^{(k)}.\alpha^{(k)}t \text{ for all } (\sigma, \mathit{is}, a : \sigma') \in \mathrm{Ran}\ \mathcal{VE}_2 \tag{8.159}$$

$$\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset \tag{8.160}$$

From Proposition 8.7.1 and from (8.160), (8.154), and (8.157), we have

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E} \text{ for all } \mathcal{E} \in \mathrm{Ran}(\mathcal{SE}_1 + \mathcal{SE}_2) \tag{8.161}$$

From (8.155), from (8.158), and from (8.160), we have

$$(\Gamma_1 + \Gamma_2)(x) = \sigma' \text{ for all } (\sigma, is, x : \sigma') \in \mathrm{Ran}(\mathcal{VE}_1 + \mathcal{VE}_2) \tag{8.162}$$

Moreover, from (8.156), from (8.159), and from (8.160), we have

$$\begin{aligned}(\Gamma_1 + \Gamma_2)(t)(c) = \sigma' \text{ and } \sigma' = \forall\alpha^{(k)}.\alpha^{(k)}t \\ \text{for all } (\sigma, is, c : \sigma') \in \mathrm{Ran}(\mathcal{VE}_1 + \mathcal{VE}_2)\end{aligned} \tag{8.163}$$

Now, from the definition of type consistency and from (8.161), (8.162), and (8.163), we have $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} (\mathcal{E}_1 + \mathcal{E}_2)$, as required. □

The preceding proposition extends to bases as follows:

Proposition 8.7.4 *If* $\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{B}_1$ *and* $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B}_2$ *and* $\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset$ *then* $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B}_1 + \mathcal{B}_2$.

PROOF Write $\mathcal{B}_1$ in the form $(\mathcal{F}_1, \mathcal{E}_1)$ and write $\mathcal{B}_2$ in the form $(\mathcal{F}_2, \mathcal{E}_2)$. From assumptions and from the definition of type consistency, we have

$$\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{E}_1 \tag{8.164}$$

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{E}_2 \tag{8.165}$$

$$\Gamma_1 \vdash_{\mathrm{tc}} \mathcal{B} \text{ for all } (\mathcal{B}, strid, strexp, \Phi) \in \mathrm{Ran}\ \mathcal{F}_1 \tag{8.166}$$

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B} \text{ for all } (\mathcal{B}, strid, strexp, \Phi) \in \mathrm{Ran}\ \mathcal{F}_2 \tag{8.167}$$

$$\mathrm{Dom}\ \Gamma_1 \cap \mathrm{Dom}\ \Gamma_2 = \emptyset \tag{8.168}$$

From Proposition 8.7.3 and from (8.164), (8.165), and (8.168), we have

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} (\mathcal{E}_1 + \mathcal{E}_2) \tag{8.169}$$

Moreover, from Proposition 8.7.2 and from (8.166), (8.167), and (8.168), we have

$$(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} \mathcal{B} \text{ for all } (\mathcal{B}, strid, strexp, \Phi) \in \mathrm{Ran}(\mathcal{F}_1 + \mathcal{F}_2) \tag{8.170}$$

Now, from the definition of type consistency and from (8.169) and (8.170), we have $(\Gamma_1 + \Gamma_2) \vdash_{\mathrm{tc}} (\mathcal{B}_1 + \mathcal{B}_2)$, as required. □

Proposition 8.7.5 *If* $\Gamma \vdash_{\mathrm{tc}} \mathcal{E}$ *and* $\mathcal{E} \gg \mathcal{E}'$ *then* $\Gamma \vdash_{\mathrm{tc}} \mathcal{E}'$.

PROOF The proof is a simple inductive argument on the depth of inference. □

The following proposition states that if a ModML declaration is translated under some assumptions $\mathcal{E}$ into an IntML declaration then the IntML declaration is typable under assumptions related to $\mathcal{E}$.

Proposition 8.7.6 (Type correctness of Core translation) *If* $\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d)$ *and* $\Gamma \vdash_{\mathrm{tc}} \mathcal{E}$ *then there exists* Γ' *such that* $\Gamma \vdash d : \Gamma'$ *and* $(\Gamma + \Gamma') \vdash_{\mathrm{tc}} \mathcal{E}'$ *and* $\mathrm{Dom}\ \Gamma' \subseteq N$. *Moreover, if* $\mathcal{E} \vdash exp \Rightarrow \tau, e$ *and* $\Gamma \vdash_{\mathrm{tc}} \mathcal{E}$ *then* $\Gamma \vdash e : \tau$.

PROOF The proof is by induction over the structure of *exp* and *dec*.

CASE $exp =$ `let` dec `in` exp `end` From assumptions and from rule 8.5, we have

$$\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d) \tag{8.171}$$
$$N \cap \mathrm{names}(\mathcal{E}, \tau) = \emptyset \tag{8.172}$$
$$\mathcal{E} + \mathcal{E}' \vdash exp' \Rightarrow \tau, e \tag{8.173}$$
$$\mathcal{E} \vdash exp \Rightarrow \tau, \texttt{let}\ d\ \texttt{in}\ e$$
$$\Gamma \vdash_{\mathrm{tc}} \mathcal{E} \tag{8.174}$$

By appropriate renaming of bound names, we can assume

$$N \cap \mathrm{names}(\Gamma, \tau) = \emptyset \tag{8.175}$$

By applying induction to (8.171) and (8.174), we have there exists Γ' such that

$$\Gamma \vdash d : \Gamma' \tag{8.176}$$
$$(\Gamma + \Gamma') \vdash_{\mathrm{tc}} \mathcal{E}' \tag{8.177}$$
$$\mathrm{Dom}\ \Gamma' \subseteq N \tag{8.178}$$

From (8.175) and from (8.178), we have

$$\mathrm{Dom}\ \Gamma \cap \mathrm{Dom}\ \Gamma' = \emptyset \tag{8.179}$$

From Proposition 8.7.3 and from (8.174), (8.177), and (8.179), we have

$$(\Gamma + \Gamma') \vdash_{\mathrm{tc}} (\mathcal{E} + \mathcal{E}') \tag{8.180}$$

By applying induction to (8.173) and (8.180), we have

$$\Gamma + \Gamma' \vdash e : \tau \tag{8.181}$$

Now, from rule 7.6 and from (8.176), (8.175), (8.178), and (8.181), we have $\Gamma \vdash \texttt{let}\ d\ \texttt{in}\ e : \tau$, as required.

CASE $exp = longvid$ From assumptions and from rule 8.1, we have

$$\mathcal{E}(longvid) = (\sigma, is, a : \sigma') \tag{8.182}$$

$$\sigma' \succ \tau \tag{8.183}$$

$$\mathcal{E} \vdash exp \Rightarrow \tau, a_\tau$$

$$\Gamma \vdash_{\text{tc}} \mathcal{E} \tag{8.184}$$

There are now two cases to consider depending on whether a is a variable or a constructor.

We first consider the case where a is a constructor c. From the definition of type consistency and from (8.184) and (8.182), we have

$$\Gamma(t)(c) = \sigma' \quad \sigma' = \forall\alpha^{(k)}.\alpha^{(k)}t \tag{8.185}$$

for some type name t. From rule 7.5, from the definition of generalisation, and from (8.183) and (8.185), we have $\Gamma \vdash c_\tau : \tau$, as required.

We now consider the case where a is a variable x. From the definition of type consistency and from (8.184) and (8.182), we have

$$\Gamma(x) = \sigma' \tag{8.186}$$

From rule 7.4 and from (8.183) and (8.186), we have $\Gamma \vdash x_\tau : \tau$, as required.

CASE $exp = \texttt{fn}^{\texttt{c}}\ longvid\ \texttt{=>}\ exp'$ From assumptions and from rule 8.3, we have

$$\mathcal{E}(longvid) = (\sigma, \mathtt{c}, c : \sigma) \tag{8.187}$$

$$\sigma \succ \tau_1 \tag{8.188}$$

$$\mathcal{E} \vdash exp' \Rightarrow \tau', e \tag{8.189}$$

$$\mathcal{E} \vdash exp \Rightarrow \tau_1 \to \tau', \lambda c : \tau_1.e$$

$$\Gamma \vdash_{\text{tc}} \mathcal{E} \tag{8.190}$$

By applying induction to (8.189) and (8.190), we have

$$\Gamma \vdash e : \tau' \tag{8.191}$$

From the definition of type consistency and from (8.190) and (8.187), we have

$$\Gamma(t)(c) = \sigma \text{ and } \sigma = \forall\alpha^{(k)}.\alpha^{(k)}t \tag{8.192}$$

for some type name t. Now, from rule 7.1, from the definition of generalisation, and from (8.188), (8.192), and (8.191), we have $\Gamma \vdash \lambda c : \tau_1.e : \tau_1 \to \tau'$, as required.

CASE $exp = \texttt{fn}^{\texttt{v}}\ vid\ \texttt{=>}\ exp'$ From assumptions and from rule 8.2, we have

$$x \notin \text{names } \mathcal{E} \tag{8.193}$$

$$\mathcal{E} + \{vid \mapsto (\tau, \texttt{v}, x : \tau)\} \vdash exp' \Rightarrow \tau', e \tag{8.194}$$

$$\mathcal{E} \vdash exp \Rightarrow \tau \to \tau', \lambda x : \tau.e$$

$$\Gamma \vdash_{\text{tc}} \mathcal{E} \tag{8.195}$$

By appropriate renaming of bound names, we can assume

$$x \notin \text{names } \Gamma \tag{8.196}$$

From the definition of type consistency, we have $(\Gamma + \{x \mapsto \tau\}) \vdash_{\text{tc}} \{x \mapsto (\tau, \texttt{v}, x : \tau)\}$, thus, from Proposition 8.7.3 and from (8.195) and (8.196), we have

$$(\Gamma + \{x \mapsto \tau\}) \vdash_{\text{tc}} (\mathcal{E} + \{x \mapsto (\tau, \texttt{v}, x : \tau)\}) \tag{8.197}$$

By applying induction to (8.194) and (8.197), we have

$$\Gamma + \{x \mapsto \tau\} \vdash e : \tau' \tag{8.198}$$

Now, from rule 7.2 and from (8.198) and (8.196), we have $\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'$, as required.

CASE $dec = \texttt{val}\ vid\ \texttt{=}\ exp$ From assumptions and from rule 8.6, we have

$$\text{tyvars } \alpha^{(k)} \cap \text{tyvars } \mathcal{E} = \emptyset \tag{8.199}$$

$$\sigma = \forall\alpha^{(k)}.\tau \tag{8.200}$$

$$\mathcal{E} \vdash exp \Rightarrow \tau, e \tag{8.201}$$

$$\mathcal{E}' = \{vid \mapsto (\sigma, \texttt{v}, x : \sigma)\} \tag{8.202}$$

$$x \notin \text{names } \mathcal{E} \tag{8.203}$$

$$\mathcal{E} \vdash dec \Rightarrow (\{x\})(\mathcal{E}', \texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e)$$

$$\Gamma \vdash_{\text{tc}} \mathcal{E} \tag{8.204}$$

By appropriate renaming of bound names and of bound type variables, we can assume

$$x \notin \text{names } \Gamma \tag{8.205}$$
$$\text{tyvars } \alpha^{(k)} \cap \text{tyvars } \Gamma = \emptyset \tag{8.206}$$

By applying induction to (8.201) and (8.204), we have

$$\Gamma \vdash e : \tau \tag{8.207}$$

Now, from rule 7.7 and from (8.205), (8.200), (8.207), and (8.206), we have

$$\Gamma \vdash \texttt{valdec}\ x : \alpha^{(k)}.\tau\ \texttt{=}\ e\ : \Gamma'$$
$$\Gamma' = \{x \mapsto \sigma\} \tag{8.208}$$

Moreover, from the definition of type consistency and from (8.202) and (8.208), we have $(\Gamma + \Gamma') \vdash_{\text{tc}} \mathcal{E}'$. We also have Dom $\Gamma' \subseteq \{x\}$, as required.

CASE $dec = \texttt{datatype}\ tyvarseq\ tycon\ \texttt{=}\ vid$ From assumptions and from rule 8.7, we have

$$tyvarseq = \alpha^{(k)} \tag{8.209}$$
$$\text{arity } t = k \tag{8.210}$$
$$\sigma = \forall\alpha^{(k)}.\alpha^{(k)}t \tag{8.211}$$
$$VE = \{vid \mapsto (\sigma, \mathtt{c})\} \tag{8.212}$$
$$\mathcal{VE} = \{vid \mapsto (\sigma, \mathtt{c}, c : \sigma)\} \tag{8.213}$$
$$\mathcal{E}' = (\{tycon \mapsto (t,\ VE)\}, \mathcal{VE}) \tag{8.214}$$
$$N = \{t, c\} \tag{8.215}$$
$$d = \texttt{datdec}\ \alpha^{(k)}t\ \texttt{=}\ c \tag{8.216}$$

By appropriate renaming of bound names, we can assume

$$t \notin \text{names } \Gamma \tag{8.217}$$

Let $\Gamma' = \{t \mapsto \{c \mapsto \sigma\}\}$. It follows from rule 7.8 and from (8.217) and (8.210) that we have $\Gamma \vdash d : \Gamma'$, as required. Moreover, from the definition of type consistency and from (8.214) and (8.212), we have $(\Gamma + \Gamma') \vdash_{\text{tc}} \mathcal{E}'$, as required. Further, from (8.215), we have Dom $\Gamma' \subseteq N$, as required.

The proofs for the remaining cases follow similarly. □

The previous proposition extends to other ModML phrases as follows:

Proposition 8.7.7 (Type correctness of Modules interpretation) *Let phrase be either a structure-level declaration, a structure-level expression, or a top-level declaration. Moreover, let $\mathcal{A}$ be either a translation environment or a translation basis. If $\mathcal{B} \vdash phrase \Rightarrow (N)(\mathcal{A}, d)$ and $\Gamma \vdash_{\text{tc}} \mathcal{B}$ then there exists Γ' such that $\Gamma \vdash d : \Gamma'$ and $(\Gamma + \Gamma') \vdash_{\text{tc}} \mathcal{A}$ and* Dom $\Gamma' \subseteq N$.

PROOF The proof is by induction over the structure of *strdec*, *strexp*, and *topdec*.

CASE $strdec = dec$ The required result follows from Proposition 8.7.6.

CASE $strdec = strdec_1\ strdec_2$ From assumptions and from rule 8.12, we have

$$\mathcal{B} \vdash strdec_1 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \tag{8.218}$$
$$(N_1 \cup N_2) \cap \text{names } \mathcal{B} = \emptyset \tag{8.219}$$
$$\mathcal{B} + \mathcal{E}_1 \vdash strdec_2 \Rightarrow (N_2)(\mathcal{E}_2, d_2) \tag{8.220}$$
$$N_2 \cap (N_1 \cup \text{names}(\mathcal{E}_1, d_1)) = \emptyset \tag{8.221}$$
$$\mathcal{B} \vdash strdec \Rightarrow (N_1 \cup N_2)(\mathcal{E}_1 + \mathcal{E}_2, d_1\ ;\ d_2)$$
$$\Gamma \vdash_{\text{tc}} \mathcal{B} \tag{8.222}$$

By appropriate renaming of bound names, we can assume

$$(N_1 \cup N_2) \cap \text{names } \Gamma = \emptyset \tag{8.223}$$

By applying induction to (8.222) and (8.218), we have there exists Γ_1 such that

$$\Gamma \vdash d_1 : \Gamma_1 \tag{8.224}$$
$$(\Gamma + \Gamma_1) \vdash_{\text{tc}} \mathcal{E}_1 \tag{8.225}$$
$$\text{Dom } \Gamma_1 \subseteq N_1 \tag{8.226}$$

By appropriate renaming of bound names, we can assume

$$N_2 \cap (N_1 \cup \text{names } \Gamma_1) = \emptyset \tag{8.227}$$

Now, from Proposition 8.7.4 and from (8.222), (8.225), (8.226), and (8.223), we have

$$(\Gamma + \Gamma_1) \vdash_{\text{tc}} (\mathcal{B} + \mathcal{E}_1) \tag{8.228}$$

By applying induction to (8.228) and (8.220), we have there exists Γ_2 such that

$$\Gamma + \Gamma_1 \vdash d_2 : \Gamma_2 \tag{8.229}$$
$$(\Gamma + \Gamma_1 + \Gamma_2) \vdash_{\text{tc}} \mathcal{E}_2 \tag{8.230}$$
$$\text{Dom } \Gamma_2 \subseteq N_2 \tag{8.231}$$

From rule 7.9 and from (8.224) and (8.229), we have

$$\Gamma \vdash d_1 \ ; \ d_2 : \Gamma_1 + \Gamma_2 \tag{8.232}$$

Now, from Proposition 8.7.4 and from (8.225), (8.230), (8.226), (8.231), and (8.227), we have $(\Gamma + \Gamma_1 + \Gamma_2) \vdash_{\text{tc}} (\mathcal{E}_1 + \mathcal{E}_2)$. Moreover, from (8.226) and from (8.231), we have $\text{Dom}(\Gamma_1 + \Gamma_2) \subseteq (N_1 \cup N_2)$, as required

CASE $strexp = longstrid$ From assumptions and from rule 8.15, we have

$$\mathcal{B}(longstrid) = \mathcal{E} \tag{8.233}$$
$$\mathcal{B} \vdash strexp \Rightarrow (\emptyset)(\mathcal{E}, \varepsilon)$$
$$\Gamma \vdash_{\text{tc}} \mathcal{B} \tag{8.234}$$

From rule 7.10, we have $\Gamma \vdash \varepsilon : \{\}$. Moreover, from the definition of type consistency and from (8.233) and (8.234), we have $\Gamma \vdash_{\text{tc}} \mathcal{E}$, as required.

CASE $strexp = strexp' : sigexp$ From assumptions and from rule 8.16, we have

$$\mathcal{B} \vdash strexp \Rightarrow (N)(\mathcal{E}, d) \tag{8.235}$$
$$\mathcal{E} \gg \mathcal{E}' \tag{8.236}$$
$$\mathcal{B} \vdash strexp \Rightarrow (N)(\mathcal{E}', d)$$
$$\Gamma \vdash_{\text{tc}} \mathcal{B} \tag{8.237}$$

By applying induction to (8.237) and (8.235), we have there exists Γ' such that $\Gamma \vdash d : \Gamma'$ and $(\Gamma + \Gamma') \vdash_{\text{tc}} \mathcal{E}$ and Dom $\Gamma' \subseteq N$. From Proposition 8.7.5 and from (8.236), we have $(\Gamma + \Gamma') \vdash_{\text{tc}} \mathcal{E}'$, as required.

CASE $strexp = funid$ ($strexp'$) From assumptions and from rule 8.17, we have

$$\mathcal{B}(funid) = (\mathcal{B}_0, strid, strexp_0, \Phi) \tag{8.238}$$

$$\mathcal{B} \vdash \mathit{strexp}' \Rightarrow (N)(\mathcal{E}, d) \tag{8.239}$$
$$\mathcal{E} \gg \mathcal{E}' \tag{8.240}$$
$$(N \cup N_1) \cap \text{names } \mathcal{B} = \emptyset \tag{8.241}$$
$$N_1 \cap (N \cup \text{names}(\mathcal{E}, d)) = \emptyset \tag{8.242}$$
$$\mathcal{B}_0 + \{\mathit{strid} \mapsto \mathcal{E}'\} \vdash \mathit{strexp}_0 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \tag{8.243}$$
$$\mathcal{B} \vdash \mathit{strexp} \Rightarrow (N \cup N_1)(\mathcal{E}_1, d \ ; \ d_1)$$
$$\Gamma \vdash_{\text{tc}} \mathcal{B} \tag{8.244}$$

By appropriate renaming of bound names, we can assume

$$(N \cup N_1) \cap \text{names } \Gamma = \emptyset \tag{8.245}$$

By applying induction to (8.244) and (8.239), we have there exists Γ' such that

$$\Gamma \vdash d : \Gamma' \tag{8.246}$$
$$(\Gamma + \Gamma') \vdash_{\text{tc}} \mathcal{E} \tag{8.247}$$
$$\text{Dom } \Gamma' \subseteq N \tag{8.248}$$

By appropriate renaming of bound names, we can assume

$$N_1 \cap (N \cup \text{names } \Gamma') = \emptyset \tag{8.249}$$

From Proposition 8.7.5, from (8.247) and (8.240), and from the definition of type consistency, we have

$$(\Gamma + \Gamma') \vdash_{\text{tc}} \{\mathit{strid} \mapsto \mathcal{E}'\} \tag{8.250}$$

From the definition of type consistency and from (8.244) and (8.238), we have

$$\Gamma \vdash_{\text{tc}} \mathcal{B}_0 \tag{8.251}$$

From (8.248) and (8.245), we have Dom $\Gamma' \cap$ Dom $\Gamma = \emptyset$, hence, from Proposition 8.7.4 and from (8.251) and (8.250), we have

$$(\Gamma + \Gamma') \vdash_{\text{tc}} (\mathcal{B}_0 + \{\mathit{strid} \mapsto \mathcal{E}'\}) \tag{8.252}$$

By applying induction to (8.243) and (8.252), we have there exists Γ_1 such that

$$\Gamma + \Gamma' \vdash d_1 : \Gamma_1 \tag{8.253}$$
$$(\Gamma + \Gamma' + \Gamma_1) \vdash_{\text{tc}} \mathcal{E}_1$$
$$\text{Dom } \Gamma_1 \subseteq N_1 \tag{8.254}$$

Now, from rule 7.9 and from (8.246) and (8.253), we have $\Gamma \vdash d \; ; \; d_1 : \Gamma' + \Gamma_1$. Moreover, from (8.248) and from (8.254), we have $\text{Dom}(\Gamma' + \Gamma_1) \subseteq (N \cup N_1)$, as required.

CASE *topdec* = `functor` *funid* `(` *strid* `:` *sigexp* `)` `=` *strexp* From assumptions and from rule 8.19, we have

$$\mathcal{F} = \{funid \mapsto (\mathcal{B}, strid, strexp, \Phi)\} \tag{8.255}$$
$$\mathcal{B} \vdash topdec \Rightarrow (\emptyset)((\mathcal{F}, \{\}), \varepsilon)$$
$$\Gamma \vdash_{\text{tc}} \mathcal{B} \tag{8.256}$$

From rule 7.10, we have $\Gamma \vdash \varepsilon : \{\}$. Moreover, from the definition of type consistency and from (8.256) and (8.255), we have $\Gamma \vdash_{\text{tc}} (\mathcal{F}, \{\})$, as required.

CASE *topdec* = $topdec_1$ $topdec_2$ The proof for this case is similar to the proof for the case for $strdec = strdec_1 \; strdec_2$.

The proofs for the remaining cases follow either immediately or immediately by induction. □

8.8 ModML Type Soundness

We now state a general type soundness result for ModML. The meaning of ModML phrases is given in terms of the interpretation into IntML declarations. Informally, the main proposition states that if a ModML top-level declaration elaborates according to the rules of the static semantics of ModML then the top-level declaration (with transparent signature constraints substituted for opaque ones) translates into an IntML declaration that does not go wrong according to the evaluation rules of IntML. The proposition does not suggest that there always exist a dynamic environment to which the IntML declaration evaluates; with appropriate support for recursive functions, a well-typed IntML declaration may fail to terminate.

Proposition 8.8.1 (ModML type soundness) *If* $B \vdash topdec \Rightarrow (T)B'$ *and* $B \succeq \overline{\mathcal{B}}$ *and* $\vdash \mathcal{B}$ *and* $\Gamma \vdash_{\mathrm{tc}} \mathcal{B}$ *and* Θ *is closed and* $\Theta \models \mathcal{D} : \Gamma$ *then there exist* $(T')B''$ *and* $(N)(\mathcal{B}', d)$ *and* Γ' *and* Θ' *such that*

- $(T)B' \succeq (T')B''$
- $\mathcal{B} \vdash oe(topdec) \Rightarrow (N)(\mathcal{B}', d)$ *and* $N \supseteq T'$ *and* $\vdash \mathcal{B}'$ *and* $\overline{\mathcal{B}'} = B''$
- $(\Gamma + \Gamma') \vdash_{\mathrm{tc}} \mathcal{B}'$ *and* $\mathrm{Dom}\ \Gamma' \subseteq N$
- $\Gamma \vdash d : \Gamma'$ *and* $\Theta' = \Theta$ of Γ'
- *if there exists* ϱ *such that* $\mathcal{D} \vdash er(d) \rightsquigarrow \varrho$ *then* $\varrho \neq wrong$ *and there exists* Θ'' *such that* $(\Theta' + \Theta'')$ *is closed and* $\mathrm{Dom}(\Theta + \Theta') \cap \mathrm{Dom}\ \Theta'' = \emptyset$ *and* $(\Theta + \Theta' + \Theta'') \models \varrho : \Gamma'$

PROOF From assumptions and from Proposition 5.3.1, we have that there exists $(T')B''$ such that

$$\overline{\mathcal{B}} \vdash oe(topdec) \Rightarrow (T')B'' \tag{8.257}$$

and $(T)B' \succeq (T')B''$, as required. From assumptions, from (8.257), and from Proposition 8.6.4, we have that there exists $(N)(\mathcal{B}', d)$ such that

$$\mathcal{B} \vdash oe(topdec) \Rightarrow (N)(\mathcal{B}', d) \tag{8.258}$$

and $N \supseteq T'$ and $\vdash \mathcal{B}'$ and $\overline{\mathcal{B}'} = B''$, as required. Moreover, from assumptions, from Proposition 8.7.7, and from (8.258), we have that there exists Γ' such that

$$\Gamma \vdash d : \Gamma' \tag{8.259}$$

and $(\Gamma + \Gamma') \vdash_{\mathrm{tc}} \mathcal{B}'$ and $\mathrm{Dom}\ \Gamma' \subseteq N$, as required. Let $\Theta' = \Theta$ of Γ'. By appropriate renaming of bound names, we can assume

$$\mathrm{Dom}\ \Theta \cap \mathrm{Dom}\ \Theta' = \emptyset \tag{8.260}$$

Now, assume that there exists a declaration result ϱ such that

$$\mathcal{D} \vdash er(d) \rightsquigarrow \varrho \tag{8.261}$$

From assumptions, from Proposition 7.4.3, and from (8.259), (8.260), and (8.261), we have $\varrho \neq \mathit{wrong}$ and there exists Θ'' such that $(\Theta' + \Theta'')$ is closed and $\mathrm{Dom}(\Theta + \Theta') \cap \mathrm{Dom}\ \Theta'' = \emptyset$ and $(\Theta + \Theta' + \Theta'') \models \varrho : \Gamma'$, as required. □

Type soundness for closed ModML top-level declarations follows from the preceding proposition:

Corollary 8.8.2 (ModML weak type soundness) *If* $\{\} \vdash \mathit{topdec} \Rightarrow (T)B$ *then there exist* $(N)(\mathcal{B}, d)$ *such that*

- $\{\} \vdash \mathit{oe}(\mathit{topdec}) \Rightarrow (N)(\mathcal{B}, d)$
- *if there exists* ϱ *such that* $\{\} \vdash \mathit{er}(d) \rightsquigarrow \varrho$ *then* $\varrho \neq \mathit{wrong}$

Similar to the approach we take here, Harper and Stone [HS97] interpret Standard ML phrases into an intermediate language for which a type soundness result exists. Their interpretation, however, interprets Modules language constructs of Standard ML into constructs of their intermediate language. In contrast, the approach that we present here eliminates all Modules language constructs during interpretation.

8.9 Cut-Off Incremental Recompilation

The framework for separate compilation presented in Chapter 6 does not immediately work well together with the interpretation of modules that we have presented in this chapter. The problem is the phase distinction between the point at which a functor is declared and the points at which the body of the functor is interpreted. Consider a project consisting of three program units `A`, `B`, and `C`:

```
(* Program unit A *)   functor A( ... ) = ...

(* Program unit B *)   functor B( ... ) = ...

(* Program unit C *)   structure A = A( ... )
                       structure B = B( ... )
                       ...
```

It is important that a modification of the functor `A`, say, does not necessarily trigger recompilation of the functor body for the functor `B` (or vice versa). However, with any reasonable definition of strong enrichment for translation environments and translation bases, a modification of program unit `A` will enforce the program unit `C` to be recompiled. Thus, according to the interpretation of Modules that we have presented in this chapter, both functor bodies will be recompiled.

The solution is to extend repositories to map functor identifiers to information about compiled functor bodies. Then, for the example above, when the program unit `C` is recompiled, it becomes possible to recognise (using strong enrichment) that the code generated for the body of the functor `B` may be reused.

Part III

The ML Kit with Regions

Chapter 9

A Guided Tour

The ML Kit with Regions (or just the Kit) is a compiler for Standard ML. Traditional implementations of Standard ML (and of most other functional languages that support dynamic data structures such as lists and trees) are based on reference tracing garbage collection, where allocation of memory is separated from deallocation of memory; the Kit is not. Instead, the Kit uses region inference [TT97, TT94] and region representation analyses [BTV96] to insert memory management directives in the program at compile time for allocating and deallocating memory.

The Kit elaborates and compiles full Standard ML—including Modules—using the techniques that we presented in the preceding parts. The Kit features, among other things, a profiler for determining space consumptions of programs, a foreign language interface for interacting with C, and a users manual that describes how to program with regions in the Kit [TBE+98].

The separate compilation framework that the Kit uses makes it possible to compile large programs using region inference. AnnoDomini, which is a tool for solving year 2000 problems in COBOL programs, has been compiled with the Kit. AnnoDomini is approximately 33,000 lines of Standard ML (excluding the Standard ML Basis Library [Ge]) and uses Standard ML Modules extensively. Moreover, the Kit compiles the Kit itself, which is approximately 80,000 lines of functorised Standard ML.

In the following section, we give an overview of how to compile programs with the Kit using a so-called project manager, which is built into the Kit (consult [TBE+98] for an in depth description.) Then, in the next sections, we describe the overall structure of the Kit. In Chapter 10, we give an overview of the back-end phases of the Kit and we show how these phases

```
import baslib.pm
in set.sml elem_int.sml main.sml
end
```

Figure 9.1: A sample project file `myprj.pm`.

provide support for the separate compilation framework presented in Chapter 6. As a running example, we demonstrate how a program is compiled with the Kit and how the program gradually is transformed into executable machine code.

9.1 Compiling with the Kit

A program in the Kit is specified by a *project file*, which lists a sequence of program units (source files) and potentially imports other project files. An example project file `myprj.pm` is listed in Figure 9.1. The meaning of the project `myprj.pm` is the meaning of the Standard ML top-level declaration that arises by catenating the program units in project `baslib.pm` and the program units `set.sml`, `elem_int.sml`, and `main.sml`. If the project `myprj.pm` is imported into other projects, then only the declarations in the program units `set.sml`, `elem_int.sml`, and `main.sml` are made visible to these other projects. Projects must be acyclic and project file names (e.g., `myprj.pm`) must be unique. A project is evaluated only once, even if it is imported by multiple other projects. Thus, if the project `myprj.pm` is imported by multiple projects then only the first import—determined by a left-to-right depth-first search of the project hierarchy—gives rise to evaluation of `set.sml`, `elem_int.sml`, and `main.sml`.

The project model that the Kit uses generalises the project model of Section 6.5 by allowing a project to import other projects. The project model provides similar functionality as the group model of the compilation manager for the Standard ML of New Jersey compiler [Blu97, Blu95]. For example, the project model allows for solving conflicts when several libraries are combined (each of them being a project) by inserting appropriate stub code for renaming problematic identifiers around a library without actually modifying the library. In contrast to the Standard ML of New Jersey compiler, the Kit

does not impose restrictions on the form of top-level declarations [Blu97].

The Kit has a simple text based user interface. It provides functionality for setting compiler flags and for managing projects. The functionality for managing projects is sparse. One operation allows the user to modify the name of the top-most project (the root project) and another operation allows the user to build a project. The Kit compiles the program units of a project in the order determined by a left-to-right depth-first traversal of the project hierarchy. Upon modification of a program unit, the Kit uses time stamps to infer what program units must be recompiled. The separate compilation scheme has the property that one always gets the same result as if the project was compiled from scratch.

Continuing the example, we assume that the project `baslib.pm` provides appropriate declarations for the identifiers `Int.toString`, `^`, and `print`. The program unit `set.sml` is listed in Figure 9.2; it declares a functor `Set`, which takes as argument a structure that provides a type `t` and a function `pr` for giving a string representation of values of the type `t`. The body of the functor declares the type `set` and operations over sets and set elements. The program unit `elem_int.sml` is listed in Figure 9.3; it declares a structure `ElemInt` with a type component `t` and a value component `pr`. The program unit `main.sml` is listed in Figure 9.4; it applies the functor `Set` to the structure `ElemInt`. Moreover, the program unit constructs a set `{5}` and prints it.

9.2 Project Management

An overview of the structure of the Kit is given in Figure 9.5. The Kit has a *project manager* that parses and analyses project hierarchies and makes requests for parsing, elaborating, and interpreting program units in the project hierarchy. When all imported projects and all program units of the project hierarchy have been processed, standard linking technology is used to link together the results of interpreting the program units in the project hierarchy. The result of the linking process is an executable file `run`.

The project manager enforces certain well-formedness requirements on project hierarchies. First, a program unit must be mentioned only once in each project file (different project files may use the same program unit name for different program units.) Second, the project hierarchy must be acyclic. Finally, different projects in the project hierarchy may be located in different directories on the underlying file system and the project model

```
functor Set ( Elem : sig eqtype t
                         val pr : t -> string
                     end ) =
  struct
    type set = Elem.t list
    val empty : set = []
    fun member (s:set, e) =
      let fun mem [] = false
            | mem (a::s) = (a = e) orelse mem s
      in mem s
      end
    fun insert (s, e) = if member(s, e) then s
                        else e::s
    fun pr s =
      let fun pr' [] = ""
            | pr' [e] = Elem.pr e
            | pr' (e::s) = Elem.pr e ^ "," ^ pr' s
      in "{" ^ pr' s ^ "}"
      end
  end
```

Figure 9.2: The program unit `set.sml`.

allows file names in projects to be given both as absolute and relative paths. Thus, to enforce that project names are unique, the project manager checks that if two projects import a project with the same name then the names refer to the same file on the underlying file system.

With reasonable assumptions about the project `baslib.pm`, the project `myprj.pm` is well-formed. To compile the project `myprj.pm`, after processing the project `baslib.pm`, the Kit processes each of the program units `set.sml`, `elim_int.sml`, and `main.sml`, in order.

```
structure ElemInt = struct type t = int
                           val pr = Int.toString
                    end
```

Figure 9.3: The program unit `elem_int.sml`.

```
structure IntSet = Set ( open ElemInt )
open IntSet
val a = insert(empty, 5)
val _ = print("The set a is " ^ pr a)
```

Figure 9.4: The program unit `main.sml`.

9.3 Parsing, Elaboration, and Opacity Elimination

The parser and lexer used in the Kit are constructed using ML-Lex [AMT94] and ML-Yacc [AT94]. A detailed description of parsing and lexing of Standard ML in the Kit is given in [BRTT93, Chapter 3].

Elaboration in the Kit is based on an implementation of algorithm W of Milner's polymorphic type discipline [Mil78, DM82] that supports efficient unification, efficient generalisation, and efficient instantiation. In the implementation, unification is based on a union find data structure and all type variables have an associated *level*; when a type variable α is unified with a type τ then the level of each of the type variables in α and τ are lowered to have the lowest level of the type variables in α and τ. Elaboration is implemented by a set of mutual recursive functions for traversing Standard ML phrases. During elaboration, a *current* level is maintained. When elaborating an expression of the form

`fn` *vid* `=>` *exp*

then *vid* is bound to a fresh type variable with its level equal to current level. When elaborating a declaration of the form

`val` *vid* `=` *exp*

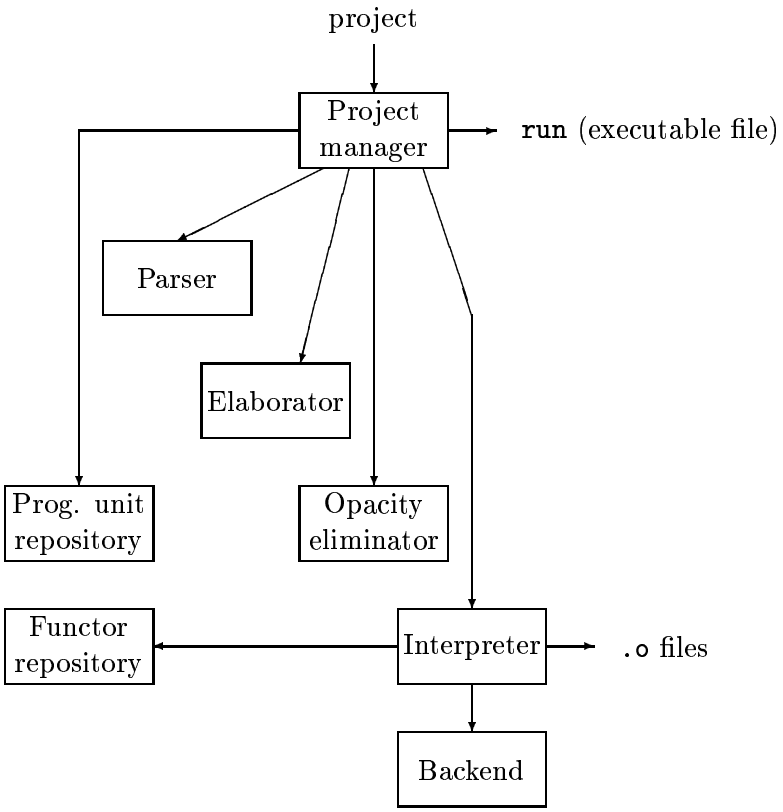

Figure 9.5: The overall structure of the Kit.

the current level is increased by one before the expression *exp* is elaborated and decreased again when returning from elaborating *exp*. Now, all the type variables that occur free in the type τ inferred for *exp* and that have level greater than the current level may be *quantified* (for full Standard ML, *exp* is also required to be what is called non-expansive [MTHM97]) by setting their level to ~1, say; all type variables in the environment will have level equal to or lower than the current level. Type *instantiation* is done by taking a copy of the type where all quantified type variables—those with level ~1—are made fresh with level equal to current level.

Many of the rules of elaboration require that some set of type names is chosen such that all of the names are fresh with respect to the environment (or

basis) in which the phrase is elaborated (see e.g., rule 2.30 on page 26). During elaboration, unification may invalidate already checked side conditions by unifying free type variables in the environment with types containing fresh type names. The Kit solves this problem by associating type variables and type names with a *rank*. During elaboration, a *current* rank is maintained; whenever a fresh type variable or a fresh type name is chosen, it gets the current rank and the current rank is increased by one. Now, when a type variable α is unified with a type τ, it is checked that the rank of each of the type names in τ is indeed lower than the rank of α. Moreover, the rank of each of the type variables in α and τ are lowered to have the lowest rank of the type variables in α and τ.

Other aspects of elaboration include how to represent realisations and how to implement signature matching. In the Kit, realisations are represented by a finite map from type names to type functions and application of a realisation to an object is implemented by applying the realisation recursively to any sub-objects of the object, while carrying out beta-conversions on types (see Section 2.3 on page 11). With this implementation of realisations and from the property that signature expressions elaborate to type-explicit signatures (see Section 3.2 on page 44), it is straightforward to implement signature matching; given a signature $(T)E$ and an environment E', the task is to find another environment E^- such that

$$(T)E \geq E^- \prec E'$$

The implementation first computes a realisation φ such that $\varphi(E) = E^-$ and Supp $\varphi \subseteq T$. Then, the implementation checks that E' indeed enriches E^-. The realisation φ is computed to be the composition of the realisations $\{t \mapsto \theta\}$, where $E(longtycon) = (t, VE)$ and $t \in T$ and $E'(longtycon) = (\theta, VE')$, for some *longtycon*, VE, and VE'.

Both during parsing and during elaboration, errors may occur, which must be reported to the programmer. Such errors are annotated on the syntax tree during parsing and elaboration and then reported to the programmer when control returns to the project manager.

During elaboration, type information is annotated on the abstract syntax tree. This type information is manipulated by opacity elimination (see Chapter 5) and used during interpretation (see Chapter 8). Thus, the Kit does not perform type inference during interpretation, but relies on type information that stems from elaborating program phrases.

9.4 Interpretation

In Chapter 8, we investigated how ModML programs can be interpreted at compile time and translated into IntML declarations. The technique used in the Kit differs slightly from the technique that we developed in Chapter 8; the Kit does not always link declarations of the intermediate language during interpretation. Instead, whenever a Core Standard ML declaration is translated into an intermediate language declaration, the Kit requests the back-end to compile these intermediate declarations into machine code object files that may later be linked using standard linking technology. For optimisation, Standard ML Modules phrases that do not contain any functor applications are compiled together to make each compilable chunk as large as possible without destroying the properties of separate compilation. To request the back-end to compile intermediate declarations into machine code object files during interpretation, the interpretation maintains *compiler bases*, which hold environments for each phase in the back-end, as well as translation bases.

Each of the program units `set.sml`, `elem_int.sml`, and `main.sml`, of the `myprj.pm` project, is first processed by the parser, the elaborator, and the opacity eliminator. As mentioned earlier, the Kit aborts the processing of the project and reports to the user if errors occur during parsing or during elaboration. After opacity elimination, the resulting representation of the program unit is processed by the interpreter. Interpretation of the `set.sml` program unit causes no code to be generated. Instead, a functor closure is associated with the functor identifier `Set`, so that, when the functor is applied, the body of the functor can be specialised for the particular application. Interpretation of the `elem_int.sml` program unit, on the other hand, requests the back-end for generating object code for the value component `pr` of the structure `ElemInt`. The optimiser in the Kit does not currently propagate simple variable bindings across program unit boundaries; if it did, no code needed be generated for the structure `ElemInt`, as all information about it (e.g., that the value identifier `pr` is bound to `Int.toString`), would be present in the resulting translation and compiler bases. Interpretation of the program unit `main.sml` requests the back-end to generate code for an instance of the body of the functor `Set` and for the construction and printing of the set `{5}`. Thus, besides from object code associated with the project `baslib.pm`, the project `myprj.pm` compiles into three separate pieces of object code, which the Kit links to form an executable `run`.

```
structure ElemInt = struct type t = int
                           fun pr a = Int.toString a
                    end
```

Figure 9.6: The program unit `elem_int.sml` after modification.

When compiling projects with heavily functorised code, it is important that functor closures in translation bases do not take up too much space. Recall that a functor closure is a record holding among its components an elaborated structure-level expression; the Kit holds on to just enough information that the structure-level expression can be reparsed and reelaborated at the points the functor is applied. The Kit does not store the abstract syntax tree itself.

9.5 The Repository

The Kit uses a *repository* for storing information about compiled program units and compiled functor bodies.

Before the manager requests the parser and elaborator to process a program unit, the manager looks up the program unit in the repository to see if information stored in the repository may be used instead of processing the program unit. Several conditions must be satisfied for the information in the repository to be used. First, the modification time for the program unit must not be newer than when the program unit was last processed.[1] Second, the assumptions under which the program unit was last parsed, elaborated, interpreted, and compiled—by the back-end—must not have changed; this last check is performed by checking for strong enrichment (for elaboration, also agreement (see Section 4.4) must be checked.) Now, consider modifying the program unit `elem_int.sml` by eta-converting the declaration of `pr` (see Figure 9.6). Then, the program unit `elem_int.sml` need be recompiled. Moreover, elaboration and compilation of the program unit `main.sml` depend on assumptions about the value component `pr` of the structure `ElemInt`. Al-

[1] An alternative is to use cryptographic checksums of program units, which can be more reliable in circumstances where times are not reliable (e.g., a distributed setting, or compilation times below one second.)

though assumptions for elaboration (i.e., the type of `pr`) have not changed, the calling convention in the Kit for `pr` has; the value component `pr` is now what is called region polymorphic and takes a region as argument at runtime (see Section 10.4).[2] It follows that the program unit `main.sml` and the body of the functor `Set` need be recompiled so as to generate correct code for the call to the `pr` value component of the structure `ElemInt`.

Similar to as when the manager makes requests to the repository, when the interpreter encounters a functor application, the interpreter requests the repository to see if information stored in the repository may be used instead of processing the functor body. Again, several conditions must be satisfied for the information in the repository to be used. First, the functor body found in the functor closure in the translation environment must not have changed. Second, the assumptions under which the functor body was last interpreted and compiled must not have changed; again, this last check is performed by checking for strong enrichment. Now, consider modifying the program unit `main.sml` to construct a larger set `{5,7}` by modifying the declaration of `a` to

```
val a = insert(insert(empty,7),5)
```

Upon this modification, the program unit `main.sml` needs be reelaborated and recompiled. However, neither the program unit `elem_int.sml` nor the body of the functor `Set` need be reelaborated or recompiled.

In the presentation of interpretation in Chapter 8, generativity of names was ensured by alpha-conversion of bound names. In an implementation that builds on standard linking technology, bound names of objects containing compiled machine code cannot be allowed to alpha-vary, because some of these bound names are labels in the compiled machine code, which are not easy to alter. Indeed, the Kit does not allow bound names of objects containing compiled machine code to alpha-vary, except at the point just before the assembly code is emitted to a file and assembled. At this point, the Kit allows bound names of the object to be renamed to potentially *match* environment entries of previously compiling the program unit or functor body. Thus matching is the key operation for implementing cut-off incremental recompilation in the Kit. Matching is defined for all phases of the Kit, for which names may be generated. These phases include elaboration, interpretation, and several phases of the back-end.

[2] We assume here that the value component `toString` of the structure `Int` is already region polymorphic.

9.6 The Back-End

In the Kit, the interpreter is parameterised over a simple interface to the back-end. The interface provides an abstract notion of *compiler basis* with operations for modification, enrichment, restriction, and matching. Moreover, it provides a function for compiling a language corresponding to IntML in some compiler basis into a compiler basis that holds information about new declarations and into so-called *target code*, which essentially is a representation of assembler code that can be emitted and assembled into a `.o` file using UNIX `as`. The interface also provides an operation for emitting target code into a `.o` file and for creating a `.o` file for executing initialisation code for all program units in a project hierarchy. As discussed earlier, standard linking technology (i.e., UNIX `ld`) is used to link `.o` files to create executables.

In the following chapter, we give an overview of the different phases of the back-end. Moreover, we describe how these phases are composed to make up the back-end.

Chapter 10

Back-End Phases

In this chapter, we give an overview of the back-end phases of the Kit and describe how each of the phases fit into the framework for separate compilation that we presented in Chapter 6 and Chapter 8. As a running example, we show how the `member` function and the `insert` function of the previous chapter are compiled into RISC like machine code.

The back-end of the Kit comprises a series of transformations on four intermediate languages of which the first three are typed:

Lambda A lambda-calculus like intermediate language corresponding to the language IntML of Chapter 7. The main difference between the Standard ML Core language and *Lambda* is that *Lambda* is typed and that it has only primitive patterns.

RegionExp The analysis that decides when regions should be allocated and deallocated is called *region inference*. Region inference inserts several forms of memory management directives into the program. *RegionExp* is the target language of region inference.

MulExp This language is similar to *RegionExp* except that the terms of *MulExp* are polymorphic in the type of information that annotate the nodes of the terms. Thus, the Standard ML type discipline allows for *MulExp* to be used as a common intermediate language for a series of the intermediate analyses of the back-end, which add more and more information on the syntax tree.

Kit Abstract Machine (KAM, for short) The KAM is a RISC like language

that details the memory management directives of *RegionExp* and *MulExp*.

In the following sections, we describe the phases of the back-end in detail. Each of the phases provides an interface with operations on abstract translation environments for modification, strong enrichment, restriction, and matching (see Chapter 6 for an explanation of these terms.) Moreover, each interface provides a function for translating a source program of the translation, in some translation environment, into a target program for the translation, together with an environment that holds information about declared names of the source program. Each of the translation phases of the back-end enjoys the properties of Chapter 6 that must hold so as to demonstrate correctness of the separate compilation framework.

The compiler basis provided by the back-end is the product of the translation environments for each of the phases of the back-end and the operations for modification, enrichment, restriction, and matching are composed, in the natural way, from the operations for each of the translation phases of the back-end.

10.1 Elimination of Polymorphic Equality

The first phase of the back-end is elimination of polymorphic equality. Standard ML has a generic primitive for testing two values of the same type for equality. Thus, the Standard ML programmer may write

```
fun eq_pair a = (3, "hello") = a
```

to create a function that takes a pair of an integer and a string as argument and tests if it is equal to the pair `(3, "hello")`; that is, the function returns the value `true` if the first component of the argument is equal to `3` and if the second component of the argument is equal to `"hello"`; otherwise, the function returns `false`. To have a primitive in the runtime system to perform this equality test, the equality primitive must be able to distinguish pairs from integers and integers from strings, and so on, so as to choose the correct means of equality; thus, traditional implementations of Standard ML use runtime tags to implement the equality primitive. In many implementations, runtime tags are already there to support reference tracing garbage collection. Because the Kit does not use reference tracing garbage collection, the runtime

tags would only be used to support an equality primitive in the runtime system.

For the preceding example, the compiler could, in principle, create code for the equality test from the knowledge of the type of the argument to the equality primitive. But such code is not always straightforward to construct. To avoid the possibility of testing functional values for equality, the type system of Standard ML [MTHM97] distinguishes between ordinary type variables, which may be instantiated to any type, and equality type variables, which may be instantiated only to types that admit equality (i.e., types not containing ordinary type variables or function types.) Now, the Standard ML programmer may declare a polymorphic function `pmem` using the equality primitive to test if a given value is among the elements of a list:

```
fun pmem y [] = false
  | pmem y (x::xs) = (y=x) orelse pmem y xs
```

The function `pmem` gets type scheme

$$\forall \varepsilon \ . \ \varepsilon \rightarrow \varepsilon \ \texttt{list} \rightarrow \texttt{bool}$$

where ε is an *equality type variable* (i.e., a type variable that ranges over equality types.) For this example, it is not immediately possible to generate code for testing `y` and `x` for equality because their types are not known. The Kit implements a translation, called equality elimination, for eliminating polymorphic equality by abstracting functions over equality functions for each bound equality type variable in the type scheme for the function. Thus, the function `pmem` translates into the function

```
fun pmem eq y [] = false
  | pmem eq y (x::xs) = eq(y,x) orelse pmem eq y xs
```

and whenever the function `pmem` is applied, an equality function for the instance of the equality type variable is passed to `pmem`. In this way, the source language for equality elimination is the language *Lambda* with a polymorphic equality primitive and the target language is the language *Lambda* without a polymorphic equality primitive. For an in-depth treatment of the translation, consult [Els98].

For each declared datatype that admits equality (i.e., that contains only types that admit equality), the Kit generates a function for testing values of this datatype for equality. Such functions are bound to *Lambda* variables

that are chosen fresh during translation and the translation maintains a *translation environment* that maps type names to the generated *Lambda* variables for equality functions for datatypes. The interface for the translation provides operations on translation environments for modification, enrichment, restriction, and matching.

10.2 Intermediate Language Optimisation

The Kit has an optimiser that transforms a *Lambda* program into another *Lambda* program by applying a series of optimisations inspired by [App92, SW95] but extended to apply to the typed setting of the *Lambda* language (see [TMC+96, Tar96]) and implemented using the techniques described in [AJ97]. The optimisations include specialisation of recursive functions, function in-lining, constant propagation, dead code elimination, minimisation of mutually recursive functions, and record elimination. All of the optimisations are local to each *Lambda* program, thus, no translation environment is used for propagating information across program unit boundaries; the Kit has yet to employ the possibilities of the separate compilation scheme with respect to intermodule optimisation. Most promising is the possibility of constant propagation and specialisation and in-lining of small functions across program unit boundaries.

After equality elimination and optimisation, the *Lambda* program resulting from interpreting the application of the `Set` functor has transformed into a new *Lambda* program. The `member` function declared by this *Lambda* program is shown in Figure 10.1. Selections from tuples (e.g., `#1 v1247` in line `(*1*)` of Figure 10.1) and deconstruction of constructed values (e.g., `decon_:: var23` in line `(*2*)` of Figure 10.1) are made explicit. Moreover, equality elimination has transformed the use of the polymorphic equality primitive into a use of the integer equality primitive (line `(*3*)` of Figure 10.1). The `insert` function of the *Lambda* program is shown in Figure 10.2.

10.3 Intermediate Language Type Checking

After a *Lambda* program is optimised it is type checked. In normal circumstances, type checking should always succeed; the phase is there to catch

```
   fun member v1247 =
(*1*) let val e = #1 v1247
          val s = #0 v1247
          fun mem var23 =
             case var23
               of nil => false
(*2*)           | :: => let val s = #1 (decon_:: var23)
                            val a = #0 (decon_:: var23)
(*3*)                   in case a =_int e
                             of true => true
                              | false => mem s
                        end
      in mem s
      end
```

Figure 10.1: The `member` function of the *Lambda* program resulting from interpreting the application of the `Set` functor (after equality elimination and optimisation). Type information has been deleted, for brevity.

errors in the generated code and it is a great help during development of the front-end and the optimisation phases. Type checking is the identity translation on *Lambda* programs, but it maintains a type environment for propagating information about declared lambda variables and declared type names to other program units that use them. The interface for the type checker provides operations for modification, enrichment, and restriction. Because no new names are generated during type checking, no matching operation is necessary for this phase in the back-end.

10.4 Region Inference

The remaining phases of the Kit are based on region inference. The basic runtime model for region based memory management is a stack of *regions* (see Figure 10.3); new regions may be allocated on top of the stack and memory may be freed either by deallocating the top most region or by *resetting* any region on the stack, without actually deallocating the region. Each region

```
fun insert v1266 =
   let val e = #1 v1266
       val s = #0 v1266
   in case member (s,e)
        of true => s
         | false => let val v1271 = (e,s)
                    in :: v1271
                    end
   end
```

Figure 10.2: The `insert` function of the *Lambda* program resulting from interpreting the application of the `Set` functor (after equality elimination and optimisation). Type information has been deleted, for brevity.

is represented as a linked list of constant sized *region pages*,[1] thus, a region may grow dynamically by new region pages being added to the region; so one may think of the region stack as a stack of heaps. Whenever a region is deallocated or a region is reset, the linked list of region pages in the region is appended to a *free list* maintained by the runtime system.

All values, such as records, recursive data structures, and closures, that do not fit into one machine word, are represented boxed and put in regions. Region inference in the Kit translates *Lambda* programs into *RegionExp* programs, for which all value creating program points in the program are annotated with region *variables*, which we range over by ρ. Thus, the *Lambda* pair `(3, "hello")` translates into the *RegionExp* pair

$$\texttt{(3, "hello" at } \rho\texttt{) at } \rho'$$

for some region variables ρ and ρ'. Moreover, when ρ is any region variable and e is some *RegionExp* expression then so is

$$\texttt{letregion } \rho \texttt{ in } e \texttt{ end}$$

Dynamically, a region is allocated on the stack of regions and bound to ρ. Then, the expression e is evaluated, perhaps using the region bound to ρ for

[1] Currently, the size of a region page in the Kit is 800 bytes.

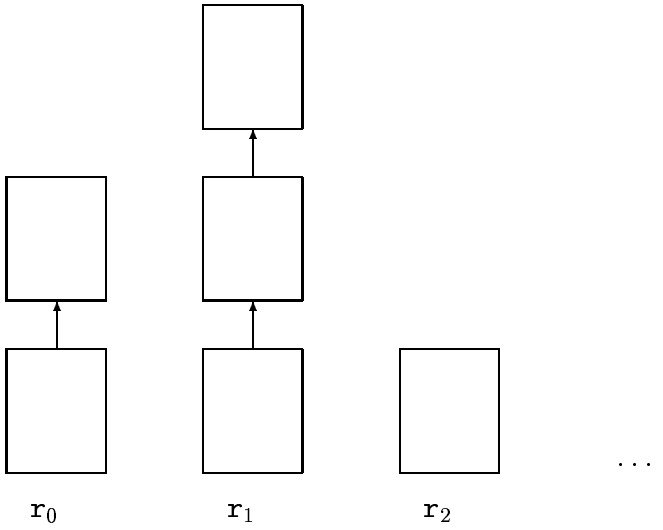

Figure 10.3: A stack of Regions. The region stack grows horizontally, thus, $\mathtt{r}_2$ is the top region. Each region is represented as a linked list of region pages.

storing values, and finally, upon reaching `end`, the region is deallocated from the stack.

A function in *RegionExp* can be declared to take regions as arguments and may thus, depending on the actual regions that are passed to the function, produce values in different regions for each call. Such functions are said to be *region polymorphic* and take the form

$$\texttt{fun}\ \mathit{vid}\ \texttt{at}\ \rho\ [\vec{\rho}]\ \mathit{arg}\ \texttt{=}\ e$$

where e is a *RegionExp* expression, where *id* and *arg* are lambda variables, where ρ is a region variable representing a region for storing the closure for the function (i.e., values for the free lambda variables for the function and regions for the free region variables for the function), and where $\vec{\rho}$ is a sequence of formal region variables for the function. When a function declaration has no free variables, no closure is needed for the function; we shall drop the `at` ρ in this case.

Region inference is a type-based analysis; if some lambda variable is bound to the *RegionExp* pair given in the preceding paragraph, then this

lambda variable gets the so-called region type

$$(t_{\text{int}} \times (t_{\text{string}}, \rho), \rho')$$

where t_{int} and t_{string} are type names with arity 0. Moreover, a region polymorphic function is associated to a so-called region type scheme, which expresses the region polymorphism of the function. Because region inference depends on region types and region type schemes for those lambda variables that occur free in *RegionExp* expressions, region inference is implemented by use of a translation environment mapping lambda variables to region types and region type schemes.

To allow for declared lambda variables in a *RegionExp* program unit to be accessed by other program units, the region inference interface in the Kit provides operations on translation environments for modification, enrichment, and restriction. No new free region variables occur in *RegionExp* programs, because such region variables are unified with already existing so-called *global* region variables (i.e., region variables that are global to the computation of the entire program.) Because no new free variables occur in *RegionExp* programs or in region types and region type schemes that propagate across *RegionExp* program boundaries, no matching operation is needed for this translation step.

Tofte and Talpin [TT94, TT97] give a type system for a language corresponding to *RegionExp* and show that the type system is sound with respect to an abstract memory model for the language; that is, they show that it is safe to deallocate a region allocated by a `letregion` construct upon leaving its scope. Tofte and Birkedal [TB96, TB98] present the region inference algorithm that is used in the Kit and demonstrate that the algorithm terminates and that the resulting program is typable in the Tofte-Talpin type system.

10.5 Region Representation Analyses

The Kit uses a series of transformations on *MulExp* programs to map the abstract memory model of the *RegionExp* language into conventional memory [BTV96]. For each translation phase, information is propagated across *MulExp* program units through translation environments, as described for region inference in the previous section.

The first transformation on *MulExp* programs is *multiplicity inference* [Vej94, BTV96]. Multiplicity inference approximates the number of values

that are put into each region at runtime. Each `letregion` construct is transformed into a construct of the form

$$\texttt{letregion}\ \rho : m\ \texttt{in}\ e\ \texttt{end}$$

where ρ is a region variable, where e is a *MulExp* expression, and where m is a *multiplicity*, which takes the form 0, 1, or ∞. The Kit implements regions with finite multiplicity (i.e., with multiplicity 0 or 1) directly on the runtime stack. Another analysis called *physical size inference* infers the maximum physical sizes (i.e., the number of words in memory) of regions with finite multiplicity [BTV96, Section 7]; thus, eventually, each `letregion` construct is transformed into the form

$$\texttt{letregion}\ \rho : psz\ \texttt{in}\ e\ \texttt{end}$$

where *psz* is a *physical size*, which take the form ∞ or n, where $n \geq 0$ is a size (in words). Similarly, each function declaration is modified so as to express the minimal physical size of regions that may be passed to the function.

Figure 10.4 shows the `member` function of the *MulExp* program resulting from translating the *Lambda* `member` function of Figure 10.1. List constructors are implemented unboxed; the least significant bit of a list value is used to distinguish the `nil` constructor from the `::` constructor, which is represented as a pointer to an allocated pair (two words) that holds the head of the list and the tail of the list. The *MulExp* `member` function takes no region arguments (expressed by the empty region argument list `[]`) and neither does the local function `mem`. However, a closure containing the local variable e is created when `mem` is declared; the closure is put into region `r10`, which is inferred to require only one word of memory.

Figure 10.5 shows the `insert` function of the *MulExp* program resulting from translating the *Lambda* `insert`-function of Figure 10.2. The *MulExp* `insert`-function takes as argument a region in which to store—potentially—a pair holding a new set element and the list representation of the existing set of elements. A local region (i.e., region `r24`) is used for holding the argument pair for the call to the `member` function.

10.6 Code Generation

The Kit translates *MulExp* programs into KAM programs, which in turn are compiled into HP PA-RISC machine code or ANSI C [EH95]. The only

```
fun member [] v1247 =
   let val e = #1 v1247
       val s = #0 v1247
   in letregion r10:1
      in let fun mem at r10 [] var23 =
                case var23
                  of nil => false
                   | :: => let val s = #1 (decon_:: var23)
                               val a = #0 (decon_:: var23)
                           in case a =_int e
                                of true => true
                                 | false => mem[] s
                           end
         in mem[] s
         end
      end
   end
```

Figure 10.4: The `member` function of the *MulExp* program resulting from translating the *Lambda* `member` function of Figure 10.1.

transformations performed on KAM programs are dead code elimination, copy propagation, and register allocation based on graph-coloring.

After register allocation, the KAM has a finite set of general purpose caller-save registers, which we shall write `gr1` through `gr4`. The KAM has a runtime stack, pointed to by the `sp` special purpose register. The runtime stack is used for preserving live variables accross function calls (using the KAM instructions `push` and `pop`), but also for storing finite regions and for storing descriptions of non-finite regions (i.e., pointers to lists of region pages.) The KAM instruction `mov` s, d moves the source s (a register or an immediate) into the destination register d. The `ld` $r(i), d$ instruction loads the content of the memory location pointed to by r offset by i words into the destination register d. Contrary, the `st` $s, r(i)$ instruction stores the content of the source register s in the memory location pointed to by r offset by i words. The `offs` $r(i), d$ instruction computes the address pointed to by r offset by i words and stores it in the destination register d.

```
fun insert [r20:2] v1266 =
   let val e = #1 v1266
       val s = #0 v1266
   in case letregion r24:2
           in member[] (s, e) at r24
           end
        of true => s
         | false => let val v1271 = (e, s) at r20
                    in :: v1271
                    end
   end
```

Figure 10.5: The `insert` function of the *MulExp* program resulting from translating the *Lambda* `insert`-function of Figure 10.2.

Figure 10.6 shows KAM code for the `member` function resulting from translating the *MulExp* member function of Figure 10.4.[2] The KAM code with the label `l_mem` corresponds to the local function `mem` of the `member` function. The KAM code for the `mem` function expects its argument in the special purpose register `arg` and its closure, holding the value for the free lambda variable e, in the special purpose register `clo`. The result of calling the `mem` function is passed in the special purpose register `res`; the KAM instruction `ret` pops the return address off the stack and does a branch. The KAM instructions `ubtagcon`, `ubdecon`, and `ubcon` allow for representing `nil` and `::` unboxed, using the least significant bit of a list value to represent the value constructor (the bit is 1 for `nil` and 0 for `::`). Consult [Els98] for a general discussion of the representation of datatypes in the Kit.

Figure 10.7 shows KAM code for the `insert` function resulting from translating the *MulExp* `insert`-function of Figure 10.5. Besides from the special purpose register `arg` the `insert` function expects, in the special purpose register `arg1`, a tuple containing the actual region for potentially storing a pair

[2]The output from the Kit is slightly beautified, for brevity; instructions, labels, and registers are renamed, conditionals are expanded (by introduction of a label and an appropriate branch instruction), the syntax for offsetting registers is modified, and the `ret` instruction is introduced as an abbreviation for a `pop` instruction and a `br` instruction. Moreover, to make allocation in regions efficient, the Kit in-lines calls to `allocInf`.

```
l_mem:
   ubtagcon  arg,gr1           % get constructor tag
   bnz       gr1,l_mem_nil     % branch on non-zero (nil)
   ld        arg(1),gr2        % load s
   ld        arg(0),gr3        % load a
   ld        clo(0),gr1        % load e from closure
   beq       gr3,gr1,l_mem_eq  % branch if a = e
   mov       gr2,arg           % install argument
   br        l_mem             % recursive call to mem
l_mem_nil:
   mov       0,res             % return false
   ret
l_mem_eq:
   mov       1,res             % return true
   ret
l_member:
   ld        arg(1),gr3        % load argument
   ld        arg(0),gr2
   mov       sp,gr1            % allocate memory for pair
   offs      sp(2),sp
   st        gr3,gr1(0)        % create pair
   st        gr1,gr1(1)
   push      l_member_cont     % push return address
   mov       gr1,clo           % install closure
   mov       gr2,arg           % install argument
   br        l_mem             % call mem
l_member_cont:
   offs      sp(~2),sp         % deallocate pair
   ret
```

Figure 10.6: KAM code for the `member` function resulting from translating the *MulExp* member function of Figure 10.4.

```
l_insert:
   ld        arg(1),gr4          % load argument
   ld        arg(0),gr2
   mov       sp,gr3              % allocate memory for pair
   offs      sp(2),sp
   st        gr2,gr3(0)          % create pair
   st        gr4,gr3(1)
   push      arg1                % push live registers
   push      gr4
   push      gr2
   push      l_insert_cont       % push return address
   mov       gr3,arg             % install argument
   br        l_member            % call member
l_insert_cont:
   pop       gr3                 % pop live registers
   pop       gr2
   pop       arg1
   offs      sp(~2),sp           % deallocate pair
   bnz       res,l_nonzero       % branch on non-zero (true)
   ld        arg1(0),gr1         % load actual region
   ccall     allocInf[gr1,2,res] % maybe allocate memory
   st        gr2,res(0)          % create pair (:: value)
   st        gr3,res(1)
   ret
l_nonzero:
   mov       gr3,res             % return s
   ret
```

Figure 10.7: KAM code for the `insert` function resulting from translating the *MulExp* `insert`-function of Figure 10.5.

holding the element e and the existing element set. Both finite regions (pointers to already allocated memory on the runtime stack) and non-finite regions (pointers to region descriptors on the runtime stack) may be passed to the function. Non-finite regions are distinguished from finite regions by having their least significant bit set. The runtime system provides functionality (i.e., the C call to `allocInf`) for allocating memory in the region, provided the least significant bit of the region is set.

Chapter 11

Conclusion

We first give an overview of what have been done. We then turn to implementation issues; in particular, we discuss how the techniques in part one and part two have been applied to the ML Kit with Regions compiler [TBE+98]. Finally, we discuss future work.

11.1 Contributions

In the first part of the thesis, we presented a language called ModML. This language is a subset of the language Standard ML [MTHM97], but, its static semantics differs from that of Standard ML in that generativity of types are modelled by type abstraction rather than by a stamp-based mechanism. This difference makes it easier to demonstrate properties of ModML; in fact, some of the modifications were needed so as to demonstrate some of the more vital properties of ModML.

One of the important properties that we demonstrated for ModML is that elaboration of a program phrase in some basis depends only on those identifiers in the basis that occur free in the phrase (in a sense that we made precise in Chapter 4.) Although this property seems trivial, problems are caused by `open` declarations and value constructors in patterns. This is one example where working out the proofs forces you to think carefully about every piece of the system; in fact, it was while doing the proof of the elaboration dependence property that we realised the problem with value constructors in patterns.

In the second part of the thesis, we developed a separate compilation

framework called cut-off incremental recompilation that allows for information about declared identifiers of a program unit to propagate to other program units that depend on the program unit. Based on properties about individual translation steps of a compiler, we have shown that the framework is sound (i.e., the result of using the framework is identical to compiling the program from scratch) and complete (i.e., the result of compiling a program from scratch is the same as if the framework was used).

The inspiration for the separate compilation framework stems from the need for more information (e.g., region type schemes) to propagate to other program units, than can usually be obtained from a program unit interface; the separate compilation framework does not support cut-off (or true) separate compilation. Therefore, it is not possible to compile parametric modules (i.e., functors) separately, before the argument modules to the parametric module have been compiled. We present an interpretation of ModML (after opaque signature constraints have been eliminated) into an intermediate language called IntML. It is explained how the interpretation of ModML is combined with the separate compilation framework; not only results of compiling program units are stored in a repository for reuse, also results of compiling bodies of parametric modules are stored.

IntML enjoys a type soundness property (see Chapter 7); it is demonstrated that this property leads to type soundness for ModML (see Chapter 8), thus, the type generativity mechanism of ModML is sufficient to ensure that value constructors do not clash at run time.

11.2 Implementation

A continual inspiration for the work presented in this thesis has been the ML Kit with Regions compiler. At the time I joined the group at DIKU four years ago, only very small (and very few) programs passed through the compiler and even fewer programs did run successfully. We now have a system that compiles all of Standard ML, provides large parts of the Standard ML Basis Library [Ge], and has a framework for managing separate compilation. The separate compilation framework makes it possible to compile large programs using region inference; the 33,000 lines Standard ML program AnnoDomini, which is a tool for solving year 2000 problems in COBOL programs, has been compiled with the Kit. The Kit has even compiled itself—about 80,000 lines of functorised Standard ML. Much effort has been spent tuning algorithms

and data representations in the Kit to make compile times comparable to other implementations. Even more effort has been spent tracing bugs and achieving a stable system; it is by no means an easy task to get a complex software project like the Kit to function properly.

In large parts, the separate compilation framework and the interpretation of Modules are independent of the back-end phases in the Kit. The ML Kit with Regions version 2 [TBE+97] implements the separate compilation framework of Chapter 6 without supporting the Modules language of Standard ML. When we implemented the interpretation of Modules, no changes to the back-end phases were necessary. In this respect, the Kit is still a kit [BRTT93]; because of the heavy use of the Standard ML Modules system, it is fairly easy to either insert new back-end phases in the Kit or to substitute old back-end phases. All that is required is that new phases provide a suitable set of operations as discussed in Chapter 6 and in Chapter 10.

11.3 Future Work

There are several possibilities for future work. First, it would be interesting to investigate the possibilities for extending the type system of ModML to support higher-order Modules [MT94, Ler95]. It would also be interesting to investigate if the interpretation of Modules carry over to higher-order module languages; in particular, it is not clear whether a combination of a separate compilation system with an interpretation of a higher-order module language is feasible.

Another area for future work is the exploitation of the support for intermodule optimisation that the framework for separate compilation provides; intermodule optimisation can have major impact on performance [Blu97, Sha97].

Bibliography

[AJ97] Andrew Appel and Trevor Jim. Shrinking lambda expressions in linear time. In *Journal of Functional Programming*, 1997.

[AMT94] Andrew Appel, James Mattson, and David Tarditi. A lexical analyzer generator for Standard ML version 1.6.0. Documentation available from the net, October 1994.

[App92] Andrew Appel. *Compiling With Continuations*. Cambridge University Press, 1992.

[App93] Andrew Appel. A critique of Standard ML. In *Journal of Functional Programming*, pages 3(4):391–429, October 1993.

[AT94] Andrew Appel and David Tarditi. ML-Yacc user's manual version 2.3. Documentation available from the net, October 1994.

[ATW94] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

[Blu95] Matthias Blume. CM, a compilation manager for SML/NJ. Technical report, Princeton University, Department of Computer Science, April 1995. User Manual.

[Blu97] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. PhD thesis, Princeton University, Department of Computer Science, November 1997.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David Turner. The ML Kit version 1. Technical report, Department of Computer Science, University of Copenhagen, March 1993. An implementation of Standard ML written in Standard ML.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23rd ACM Symposium on Principles of Programming Languages*, January 1996.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In *24th ACM Symposium on Principles of Programming Languages*, January 1997.

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, January 1982.

[EH95] Martin Elsman and Niels Hallenberg. An optimizing backend for the ML Kit using a stack of regions. Student Project, July 1995.

[Els98] Martin Elsman. Polymorphic equality – no tags required. In *Second International Workshop on Types in Compilation*, March 1998.

[Ge] Emden Gansner and John Reppy (eds.). The Standard ML Basis Library reference manual. In preparation.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *21st ACM Symposium on Principles of Programming Languages*, January 1994.

[HS97] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical report, Carnegie Mellon University, June 1997. CMU-CS-97-147.

[Jon95] Mark P. Jones. From hindley-milner types to first-class structures. In *Proceedings of the 1995 Haskell Workshop, La Jolla, California*, June 1995. Yale University Research Report YALEU/DCS/RR-1075.

[Kah93] Stefan Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical report, University of Edinburgh, Laboratory for Foundations of Computer Science, April 1993. There is an Addenda for this paper, written June 94.

[Ler92] Xavier Leroy. *Polymorphic Typing of an Algorithmic Language.* PhD thesis, INRIA, October 1992.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22nd ACM Symposium on Principles of Programming Languages*, pages 142–153, 1995.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[Mac92] David MacQueen. Reflections on Standard ML. In *Programming, Concurrency, Simulation and Automated Reasoning*, pages 32–46. Springer-Verlag, LNCS 693, 1992.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. In *Journal of Computer and System Sciences*, pages 17:348–375, 1978.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, 1991.

[MT94] David MacQueen and Mads Tofte. A semantics for higher-order functors. In *Fifth European Symposium on Programming*, pages 409–423. Springer-Verlag, April 1994.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Rus98] Claudio V. Russo. *Types for Modules.* PhD thesis, University of Edinburgh, June 1998.

[SA93] Zhong Shao and Andrew Appel. Smartest recompilation. In *20th ACM Symposium on Principles of Programming Languages*, January 1993.

[Sha97] Zhong Shao. Typed cross-module compilation. Technical report, Department of Computer Science, Yale University, July 1997. YALEU/DCS/TR-1126.

[SW95] Manuel Serrano and Pierre Weis. Bigloo: a portable and optimizing compiler for strict functional languages. In *Second International Symposium on Static Analysis*, pages 366–381, September 1995.

[Tar96] David Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, December 1996.

[TB96] Mads Tofte and Lars Birkedal. Unification and polymorphism in region inference. Accepted for the Milner Festschrift (25 pages), 1996.

[TB98] Mads Tofte and Lars Birkedal. A region inference algorithm. Transactions on Programming Languages and Systems (TOPLAS) (Accepted), November 1998.

[TBE+97] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical report, Department of Computer Science, University of Copenhagen, April 1997.

[TBE+98] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit (for version 3). Technical report, Department of Computer Science, University of Copenhagen, December 1998.

[Tic86] Walter Tichy. Smart recompilation. In *ACM Transactions on Programming Languages and Systems*, pages 273–291, July 1986.

[TMC+96] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, Department of Computer Science, May 1988.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *21st ACM Symposium on Principles of Programming Languages*, January 1994.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[Vej94] Magnus Vejlstrup. Multiplicity inference. Master's thesis, Department of Computer Science, University of Copenhagen, September 1994.