Static Interpretation of Modules

Martin Elsman
mael@cs.berkeley.edu

Computer Science Division, University of California, Berkeley*

Abstract

This paper presents a technique for compiling Standard ML Modules into typed intermediate language fragments, which may be compiled separately and linked using traditional linking technology to form executable code. The technique is called *static interpretation* and allows compile-time implementation details to propagate across module boundaries. Static interpretation eliminates all module-level code at compile time.

The technique scales to full Standard ML and is used in the ML Kit with Regions compiler. A framework for smart recompilation makes the technique useful for compiling large programs.

1 Introduction

This paper describes a novel approach to the compilation of Standard ML Modules, in which the Modules language is regarded as a linking language for specifying how code fragments can be combined to form a complete program. As in most ML compilers, functors are elaborated when they are declared. Thus, type errors in functors are caught already when the functor is declared (as opposed to when it is applied). However, code generation for the functor is postponed till the functor is applied. Indeed, if a functor is applied to two different arguments, code for its body will be generated twice. We refer to this approach to module compilation as *static interpretation*.

The one most important advantage of static interpretation is that it allows compile-time information to propagate across module boundaries. When the body of a functor is compiled, the actual argument to which it is applied is known. Thus, one can make use of the information the compiler has about the actual argument when code is generated for the functor body. Region information is a case in point.

Region inference [17, 18, 1, 15] is a type based analysis for inserting allocation and deallocation directives in a program, so as to reduce or completely eliminate pointer-traversing garbage collection. The runtime heap is divided into a stack of expandable *regions*. Value-creating expressions are annotated with information that directs in what regions values go at runtime. If *exp* is some region-annotated expression, then so is

letregion r in *exp* **end**

Such expressions are evaluated as follows. First a region is allocated on the stack and bound to the *region variable* **r**. Then *exp* is evaluated, possibly using the region bound to **r** for holding values. Finally, upon reaching **end**, the region is reclaimed. Another important aspect of region based memory management is *region polymorphism*, which allows a function to use different regions for different invocations. Region inference assigns a region polymorphic function a *region type scheme*, which captures the memory behavior of the function.

Now, consider the functor declaration

```
functor F(X: sig type t
                 val f : int -> t
                 val g : t -> int * int
             end) =
  struct
    val it = #1(X.g(X.f 5))
  end;
```

The formal parameter signature for X does not give any region type schemes for f and g. For example, the signature does not say whether g creates the pair it returns in a fresh region, or perhaps always returns some fixed pair that resides in a global region. Consequently, region inference of the body of F is not possible; we cannot know whether the pair returned by g can be deallocated after the first projection has been applied to it.

Now consider an application of F:

```
structure S = F(struct type t = int
                       fun f(n:int) = n
                       fun g(n:int) = (n,n)
                end)
```

Static interpretation of the application results in two intermediate language fragments, one for the argument to F, and one for the body of F. The two fragments can be compiled into machine code separately and linked using traditional linking technology. During compilation of the second fragment, the compiler can apply the information that results from compiling the first fragment. The code for the application is equivalent to the region-annotated program

*Work done while at University of Copenhagen.

To appear in the 1999 International Conference on Functional Programming (ICFP'99), September 1999, Paris, France.

```
local
  type t = int
  fun f(n:int) = n
  fun g(n:int)[r] = (n,n) at r
in
  val it = letregion r
           in #1(g(f 5)[r])
           end
end
```

Region inference of the actual argument to the application of F shows that the actual g creates a fresh pair. Therefore, region inference determines that the pair created by g can indeed be reclaimed after the first projection has been applied to it.

Another advantage of static interpretation is that no module-level code exists at runtime (such code can be large, and often, it is executed only once.) Static interpretation performs all module-level execution during compilation. Signature matching generates no code either, not even in the case where signature matching imposes a less polymorphic type on a value in a structure. Conceptually, the Modules language is used for combining declarations of the Core language; no overhead is incurred by programming with modules.

Delaying code generation till functor application time is not feasible, unless it is integrated with a technique for smart recompilation of modules. Consider a program consisting of the three compilation units

```
unit 1:     functor f(X : ... ) = struct ... end

unit 2:     functor g(Y : ... ) = struct ... end

unit 3:     structure A = f ( ... )
            structure B = g (A)
```

Suppose that the user makes a modification to unit 1. Some separate compilation scheme will say that because unit 3 uses the functor f then unit 3 need be recompiled. And because functors are compiled at the point they are applied, recompilation of unit 3 means that the body of the functor g is recompiled as well. To explain the solution to this problem, let us consider the example once more. Because the declaration of f has changed and because f is applied in unit 3, we do need to reinterpret unit 3. Because the declaration of f has changed, the static interpretation of the application of f in unit 3 results in fresh code for the body of f. However, we might be able to avoid reinterpretation of the body of g. When code is first generated for the body of g, the environment under which g is interpreted is memorised. This environment is the combination of the compile time environment that existed when g was declared and the compile time environment for the actual argument to g (i.e., A). Now, before attempting to reinterpret the body of g, static interpretation can check if the environment has changed, and if not, the code previously generated for g can be reused. However, because type inference and compilation generate fresh names of various kinds (e.g., type names and even machine code labels), it is essential that the result of compiling the body of f the second time around is *matched* (by renaming of freshly generated names) against the result of compiling the body of f the first time.

In this paper, we formally develop the theory of static interpretation for a skeleton module language and thus provide a general solution to the problem of propagating compile-time information across module boundaries. We show that static interpretation is safe in the sense that our skeleton language is interpreted into an intermediate language that possesses a type soundness property. Moreover, by considering the meaning of the source language as being defined by the interpretation, the result is proof of type soundness of the module language. Static interpretation is used for compiling Standard ML Modules in the ML Kit with Regions compiler (the Kit), which is a compiler for full Standard ML based on region inference [16]. Experience with the Kit tells us that static interpretation scales to the compilation of large programs, such as the Kit itself, which is about 87,000 lines of Standard ML.

In the following section, we present the source language of static interpretation. In Section 3, we present the intermediate language, which is the target language for the static interpretation presented in Section 4. Opaque signature constraints are discussed in Section 5. The use of static interpretation in the Kit is discussed in Section 6. In the final sections, we describe related work and conclude.

2 The Source Language

The source language is divided into a *Core* language and a *Modules* language. Static interpretation is in large parts independent of the Core language, although the technique does depend on certain properties of this language, which in this paper are illustrated by considering a small ML like Core language.

Some modifications to the static semantics of SML'97 [11] were appropriate, so as to demonstrate important properties of static interpretation. The main modification is that type generativity is modeled by type abstraction. One essential property of the source language is that of Proposition 2.1, which guarantees that if a functor and an application of the functor can be typed, then so can the body of the functor with the formal parameter appropriately replaced by the actual argument. In contrast, there are kinds of parameterised modules that cannot be type checked in isolation, such as C++ templates.

2.1 Grammar and Syntactic Notations

We divide *identifiers* into classes VId of *value identifiers* (*vid*), TyCon of *type constructors* (*tycon*), TyVar of *type variables* (α), StrId of *structure identifiers* (*strid*), and FunId of *functor identifiers* (*funid*). For each class of identifiers, ranged over by x, there is a class of *long identifiers*, ranged over by *longx*, defined inductively as either an identifier x or a *qualified identifier strid.longx'*, for some structure identifier *strid* and long identifier *longx'*. When x ranges over objects of some class, we write $x^{(k)}$, $k \geq 0$ or $x_1 \cdots x_n$ to denote a *sequence* of k objects of this class. In program text and when $k \geq 2$, we write $(x_1, \cdots, x_k)$ to denote the sequence.

The grammars for Core and Modules are shown in Figure 1. Function type expressions associate to the right and function applications associate to the left.

Structure components can be accessed either through qualified identifiers or through the `open` declaration. Local declarations are supported by use of `let` expressions. To keep the Core language simple, datatypes can have only one value constructor that does not take arguments. This

ty ::= ty_1 -> ty_2 | α
  | $ty^{(k)}$ *longtycon*

exp ::= *longvid* | exp_1 exp_2
  | fn *longvid* => *exp*
  | let *dec* in *exp* end

dec ::= val *vid* = *exp*
  | datatype $\alpha^{(k)}$ *tycon* = *vid*
  | type $\alpha^{(k)}$ *tycon* = *ty*
  | open *longstrid*

sigexp ::= sig *spec* end | *sigexp* where type $\alpha^{(k)}$ *longtycon* = *ty*

spec ::= val *vid* : *ty* | type $\alpha^{(k)}$ *tycon* | datatype $\alpha^{(k)}$ *tycon* = *vid*
  | structure *strid* : *sigexp* | $spec_1$ $spec_2$ | ε

strexp ::= struct *strdec* end | *longstrid* | *strexp* : *sigexp* | *funid* (*strexp*)

strdec ::= *dec* | structure *strid* = *strexp* | $strdec_1$ $strdec_2$ | ε

topdec ::= *strdec* | $topdec_1$ $topdec_2$ | ε
  | functor *funid* (*strid* : *sigexp*) = *strexp*

Figure 1: Grammar for Core (left) and Modules (right).

simple form of datatypes encapsulates the features of generativity and identifier status; it is straight-forward to extend the language to support more liberal forms of datatypes [3].

2.2 Basic Semantic Objects

We assume a denumerably infinite set Name of *names* and a denumerably infinite set TyName $\subseteq$ Name of *type names* (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 $t = k$. We use N and T to range over sets of names and sets of type names, respectively. Moreover, we use NameSet to denote the set of all name sets and TyNameSet to denote the set of all type name sets.

A *(semantic) type* τ is either a function type $\tau_1 \to \tau_2$, a type variable α, or a constructed type $\tau^{(k)}t$. Because two different type constructors in the source language may denote the same type, type names are used to model the 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 $t = t'$ and, recursively, $\tau_i = \tau'_i$, $i = 1..k$.

A *type function* θ is an object of the form $\Lambda\alpha^{(k)}.\tau$, with *arity* k. Type functions must be *closed* (i.e., tyvars$(\tau) \subseteq \alpha^{(k)}$) and the bound variables must be distinct. We use TypeFcn to denote the set of type functions. 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 sequence $\tau^{(k)}$ 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.

A *substitution* S is a finite map from type variables to types. By natural extension, a substitution can be applied to any semantic object that does not bind type names; its effect is to replace each type variable α by $S(\alpha)$, with appropriate renaming of bound type variables.

A *type scheme* σ is an object of the form $\forall\alpha^{(k)}.\tau$. We use TypeSch to denote the set of all type schemes. 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 up-to renaming and reordering of bound variables, and deletion from the prefix type variables that do not occur in the body. We sometimes consider a type τ to be a type scheme, identifying it with $\forall().\tau$. It is easy to verify that generalisation of type schemes is reflexive and transitive. Moreover, generalisation is closed under substitution, that is, $\sigma \succ \sigma'$ implies $S(\sigma) \succ S(\sigma')$, for any substitution S.

2.3 Compound Semantic Objects

The compound semantic objects for elaboration are as follows:

$$
\begin{array}{rcl}
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{Env} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{TypeFcn} \times \mathrm{ValEnv} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \xrightarrow{\mathrm{fin}} \mathrm{TypeSch} \times \mathrm{IdStatus} \\
E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
\Sigma & \in & \mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Env} \\
\Phi & \in & \mathrm{FunSig} = \mathrm{TyNameSet} \times (\mathrm{Env} \times \mathrm{Sig}) \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \xrightarrow{\mathrm{fin}} \mathrm{FunSig} \\
B & \in & \mathrm{StatBasis} = \mathrm{FunEnv} \times \mathrm{Env} \\
(T)B & \in & \mathrm{ProgSig} = \mathrm{TyNameSet} \times \mathrm{Basis}
\end{array}
$$

A value environment VE associates value identifiers with a so-called *identifier status* (*is*). An identifier status can be either c (constructor status) or v (value status) and we use IdStatus to denote the set $\{\mathtt{c}, \mathtt{v}\}$.

The prefix (T) in signatures $(T)E$, functor signatures $(T)(E, \Sigma)$, and program signatures $(T)B$ binds type names. Two such objects are considered equal up-to renaming of bound names and 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.

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

2.4 Typing Rules for Core

Typing rules for the Core are given in Figure 2. The rules allow inferences among sentences of the form $E \vdash phrase \Rightarrow A$, where *phrase* is a Core language phrase, E is an environment providing assumptions for free long identifiers in *phrase*, and A is either a type or a signature. Sentences of this form are

$$\boxed{E \vdash ty \Rightarrow \tau \;\text{ and }\; E \vdash exp \Rightarrow \tau \;\text{ and }\; E \vdash dec \Rightarrow (T)E}$$

$$\frac{E \vdash ty_1 \Rightarrow \tau_1 \qquad E \vdash ty_2 \Rightarrow \tau_2}{E \vdash ty_1 \;\texttt{->}\; ty_2 \Rightarrow \tau_1 \to \tau_2} \; (1) \qquad \frac{tyvar = \alpha}{E \vdash tyvar \Rightarrow \alpha} \; (2) \qquad \frac{E(longtycon) = (\theta^{(k)}, VE) \quad E \vdash ty_i \Rightarrow \tau_i,\ i = 1..k}{E \vdash ty_1 \cdots ty_k\ longtycon \Rightarrow (\tau_1, \cdots, \tau_k)\theta^{(k)}} \; (3)$$

$$\frac{\sigma \text{ of } E(longvid) \succ \tau}{E \vdash longvid \Rightarrow \tau} \; (4) \qquad \frac{vid \notin \mathrm{Dom}\ E \;\text{ or }\; is \text{ of } E(vid) = \mathtt{v} \quad E + \{vid \mapsto (\tau, \mathtt{v})\} \vdash exp \Rightarrow \tau'}{E \vdash \mathtt{fn}\ vid\ \texttt{=>}\ exp \Rightarrow \tau \to \tau'} \; (5) \qquad \frac{E(longvid) = (\sigma, \mathtt{c}) \quad \sigma \succ \tau \quad E \vdash exp \Rightarrow \tau'}{E \vdash \mathtt{fn}\ longvid\ \texttt{=>}\ exp \Rightarrow \tau \to \tau'} \; (6)$$

$$\frac{E \vdash exp_1 \Rightarrow \tau' \to \tau \qquad E \vdash exp_2 \Rightarrow \tau'}{E \vdash exp_1\ exp_2 \Rightarrow \tau} \; (7) \qquad \frac{E \vdash dec \Rightarrow (T)E' \qquad E + E' \vdash exp \Rightarrow \tau \qquad T \cap (\mathrm{tynames}(E, \tau)) = \emptyset}{E \vdash \mathtt{let}\ dec\ \mathtt{in}\ exp\ \mathtt{end} \Rightarrow \tau} \; (8)$$

$$\frac{E \vdash exp \Rightarrow \tau \qquad \{\alpha^{(k)}\} \cap \mathrm{tyvars}\ E = \emptyset}{E \vdash \mathtt{val}\ vid\ \texttt{=}\ exp \Rightarrow (\emptyset)\{vid \mapsto (\forall\alpha^{(k)}.\tau, \mathtt{v})\}} \; (9) \qquad \frac{E \vdash ty \Rightarrow \tau}{E \vdash \mathtt{type}\ \alpha^{(k)}\ tycon\ \texttt{=}\ ty \Rightarrow (\emptyset)\{tycon \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}} \; (10)$$

$$\frac{\mathrm{arity}\ t = k \qquad VE = \{vid \mapsto (\forall\alpha^{(k)}.\alpha^{(k)}t, \mathtt{c})\}}{E \vdash \mathtt{datatype}\ \alpha^{(k)}\ tycon\ \texttt{=}\ vid \Rightarrow (\{t\})(\{tycon \mapsto (t, VE)\}, VE)} \; (11) \qquad \frac{E(longstrid) = E'}{E \vdash \mathtt{open}\ longstrid \Rightarrow (\emptyset)E'} \; (12)$$

Figure 2: Typing rules for Core.

read "*phrase* elaborates to A in E." The rules are similar to those of SML'97 with one important difference. For the typing rules of declarations, type names that are bound in the resulting signature are those type names that must be considered fresh. Such bound type names stem from datatype declarations; disjointness of bound type names from other type names is explicitly enforced in the rules. For example, without the side condition in rule 8, the typing rules become unsound [7]. In SML'97, uniqueness of local type names is enforced by a non-local requirement.

2.5 Realisations and Instantiation

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 $\varphi(t) \neq t$. Realisations φ are extended to apply to all semantic objects; their effect is to replace each type name t by $\varphi(t)$, with appropriate renaming of bound type names.

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 $\varphi \subseteq T_1$.

A prefix (T) in some object can be thought of (depending on the context) as either existentially quantifying T or universally quantifying T. For instance, a functor signature $(T)(E, (T')E')$ can be read "$\forall T.(E, \exists T'.E')$."

2.6 Enrichment and Signature Matching

Enrichment allows for hiding of components of a structure. Formally, 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 $\theta_1 = \theta_2$ and 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 $\sigma_1 \succ \sigma_2$ and 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 E' such that $\Sigma \geq E' \prec E$.

2.7 Typing Rules for Modules

The typing rules for Modules are given in Figure 3. The rules allow inferences among sentences of the form $B \vdash phrase \Rightarrow A$, where *phrase* is a Modules phrase, B is a basis, and A is either a signature or a program signature. Sentences of this form are read "*phrase* elaborates to A in B." The rules are similar to those of SML'97 with the main difference that type generativity is modeled by type abstraction. In rule 24, for instance, type generativity is enforced by appropriate alpha-conversion.

The following proposition expresses that if the body of a functor is typable under assumptions provided by the formal argument of the functor, then the body is also typable under assumptions provided by any functor instance:

Proposition 2.1 (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 The proof is based on the definition of functor signature instantiation and on the property that elaboration of structure expressions is closed under realisation [3, Chapter 8]. □

$B \vdash spec \Rightarrow \Sigma$ and $B \vdash sigexp \Rightarrow \Sigma$

$$\frac{E \text{ of } B \vdash ty \Rightarrow \tau \qquad \alpha^{(k)} = \text{tyvars}\,\tau}{B \vdash \texttt{val}\ vid\ \texttt{:}\ ty \Rightarrow (\emptyset)\{vid \mapsto (\forall\alpha^{(k)}.\tau, \texttt{v})\}} \ (13) \qquad \frac{\text{arity}\ t = k}{B \vdash \texttt{type}\ \alpha^{(k)}\ tycon \Rightarrow (\{t\})\{tycon \mapsto (t, \{\})\}} \ (14)$$

$$\frac{\text{arity}\ t = k \qquad VE = \{vid \mapsto (\forall\alpha^{(k)}.\alpha^{(k)}t, \texttt{c})\}}{B \vdash \texttt{datatype}\ \alpha^{(k)}\ tycon\ \texttt{=}\ vid \Rightarrow (\{t\})(\{tycon \mapsto (t, VE)\}, VE)} \ (15) \qquad \frac{B \vdash sigexp \Rightarrow (T)E}{B \vdash \texttt{structure}\ strid\ \texttt{:}\ sigexp \Rightarrow (T)\{strid \mapsto E\}} \ (16)$$

$$\frac{\begin{array}{c} B \vdash spec_1 \Rightarrow (T_1)E_1 \qquad (T_1 \cup T_2) \cap \text{tynames}\ B = \emptyset \\ B + E_1 \vdash spec_2 \Rightarrow (T_2)E_2 \qquad T_2 \cap (T_1 \cup \text{tynames}\ E_1) = \emptyset \qquad \text{Dom}\ E_1 \cap \text{Dom}\ E_2 = \emptyset \end{array}}{B \vdash spec_1\ spec_2 \Rightarrow (T_1 \cup T_2)(E_1 + E_2)} \ (17) \qquad \frac{}{B \vdash \varepsilon \Rightarrow (\emptyset)\{\}} \ (18)$$

$$\frac{\begin{array}{c} B \vdash sigexp \Rightarrow (T)E \qquad T \cap \text{tynames}\ B = \emptyset \\ E \text{ of } B \vdash ty \Rightarrow \tau \qquad E(longtycon) = (t, VE) \qquad t \in T \qquad \varphi = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \end{array}}{B \vdash sigexp\ \texttt{where type}\ \alpha^{(k)}\ longtycon\ \texttt{=}\ ty \Rightarrow (T)(\varphi(E))} \ (19) \qquad \frac{B \vdash spec \Rightarrow \Sigma}{B \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow \Sigma} \ (20)$$

$B \vdash strexp \Rightarrow \Sigma$ and $B \vdash strdec \Rightarrow \Sigma$

$$\frac{B \vdash strdec \Rightarrow \Sigma}{B \vdash \texttt{struct}\ strdec\ \texttt{end} \Rightarrow \Sigma} \ (21) \qquad \frac{B(longstrid) = E}{B \vdash longstrid \Rightarrow (\emptyset)E} \ (22)$$

$$\frac{\begin{array}{c} B \vdash strexp \Rightarrow (T)E \qquad B \vdash sigexp \Rightarrow \Sigma \\ \Sigma \geq E' \prec E \qquad T \cap \text{tynames}\ B = \emptyset \end{array}}{B \vdash strexp\ \texttt{:}\ sigexp \Rightarrow (T)E'} \ (23) \qquad \frac{\begin{array}{c} B \vdash strexp \Rightarrow (T)E \qquad B(funid) \geq (E'', (T')E') \\ E \succ E'' \qquad (T \cup T') \cap \text{tynames}\ B = \emptyset \end{array}}{B \vdash funid\ \texttt{(}\ strexp\ \texttt{)} \Rightarrow (T \cup T')E'} \ (24)$$

$$\frac{E \text{ of } B \vdash dec \Rightarrow \Sigma}{B \vdash dec \Rightarrow \Sigma} \ (25) \qquad \frac{B \vdash strexp \Rightarrow (T)E}{B \vdash \texttt{structure}\ strid\ \texttt{=}\ strexp \Rightarrow (T)\{strid \mapsto E\}} \ (26)$$

$$\frac{\begin{array}{c} B \vdash strdec_1 \Rightarrow (T_1)E_1 \qquad (T_1 \cup T_2) \cap \text{tynames}\ B = \emptyset \\ B + E_1 \vdash strdec_2 \Rightarrow (T_2)E_2 \qquad T_2 \cap (T_1 \cup \text{tynames}\ E_1) = \emptyset \end{array}}{B \vdash strdec_1\ strdec_2 \Rightarrow (T_1 \cup T_2)(E_1 + E_2)} \ (27) \qquad \frac{}{B \vdash \varepsilon \Rightarrow (\emptyset)\{\}} \ (28)$$

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

$$\frac{B \vdash strdec \Rightarrow (T)E}{B \vdash strdec \Rightarrow (T)E} \ (29) \qquad \frac{B \vdash sigexp \Rightarrow (T)E \qquad B + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma \qquad T \cap \text{tynames}\ B = \emptyset}{B \vdash \texttt{functor}\ funid\ \texttt{(}\ strid\ \texttt{:}\ sigexp\ \texttt{)}\ \texttt{=}\ strexp \Rightarrow (\emptyset)\{funid \mapsto (T)(E, \Sigma)\}} \ (30)$$

$$\frac{\begin{array}{c} B \vdash topdec_1 \Rightarrow (T_1)B_1 \qquad (T_1 \cup T_2) \cap \text{tynames}\ B = \emptyset \\ B + B_1 \vdash topdec_2 \Rightarrow (T_2)B_2 \qquad T_2 \cap (T_1 \cup \text{tynames}\ B_1) = \emptyset \end{array}}{B \vdash topdec_1\ topdec_2 \Rightarrow (T_1 \cup T_2)(B_1 + B_2)} \ (31) \qquad \frac{}{B \vdash \varepsilon \Rightarrow (\emptyset)\{\}} \ (32)$$

Figure 3: Typing rules for Modules.

3 An Intermediate Language

We now present the intermediate language used as the target language for static interpretation.

We assume a denumerably infinite set IVar $\subseteq$ Name of *variables* (x) and a denumerably infinite set ICon $\subseteq$ Name of *constructors* (c) . We use a to denote either a variable or a constructor. Here is the grammar for the language:

$$e \;::=\; \lambda a:\tau.e \;\mid\; e_1\; e_2 \;\mid\; a_\tau \;\mid\; \texttt{let}\; d\; \texttt{in}\; e$$

$$\begin{array}{lcl} d & ::= & \texttt{valdec}\; x : \alpha^{(k)}.\tau \;\texttt{=}\; e \\ & \mid & \texttt{datdec}\; \alpha^{(k)}t \;\texttt{=}\; c \;\mid\; d_1 \;;\; d_2 \;\mid\; \varepsilon \end{array}$$

A type name in the intermediate language is bound uniquely by a datatype declaration, which mentions the set of value constructors associated with the type name. For simplicity, the intermediate language allows datatype declarations to have only one value constructor that takes no arguments. The language can easily be extended to allow more liberal forms of datatypes.

A function $\lambda c:\tau.e$, where c is a constructor, resembles a simple pattern-match construct with only one branch. The construct is used for translating pattern-match constructs of the Core language. Dynamically, the function fails if it is applied to a value different from the constructor c. The soundness result guarantees that for well-typed phrases, the pattern-match mechanism always succeeds. Because the constructor is not considered bound within the body of the function, the constructor may not be renamed.

In contrast, the variable x in a function of the form $\lambda x:\tau.e$ is bound within its body e. Similarly, the type variables $\alpha^{(k)}$ in the value declaration $\texttt{valdec}\; x : \alpha^{(k)}.\tau \;\texttt{=}\; e$ are considered bound within τ and e. Functions and value declarations are considered equivalent up to renaming of bound variables and bound type variables. The *declared* names of a declaration d, written $decl(d)$, is defined by the equations

$$\begin{array}{rcl} decl(\texttt{datdec}\; \alpha^{(k)}t \;\texttt{=}\; c) & = & \{t\} \\ decl(\texttt{valdec}\; x : \alpha^{(k)}.\tau \;\texttt{=}\; e) & = & \{x\} \\ decl(d_1 \;;\; d_2) & = & decl(d_1) \cup decl(d_2) \\ decl(\varepsilon) & = & \emptyset \end{array}$$

For declarations of the form $d_1 \;;\; d_2$, declared names of d_1 are bound in d_2. The language supports local declarations through the use of expressions of the form $\texttt{let}\; d\; \texttt{in}\; e$. We consider such expressions equivalent up to renaming of declared names of d.

3.1 Typing Rules

The typing rules make use of the following additional semantic objects:

$$\begin{array}{lcl} \Delta & \in & \text{IVarEnv} = \text{IVar} \xrightarrow{\text{fin}} \text{TypeSch} \\ \Theta & \in & \text{ITyEnv} = \text{TyName} \xrightarrow{\text{fin}} (\text{ICon} \xrightarrow{\text{fin}} \text{TypeSch}) \\ \Gamma & \in & \text{IEnv} = \text{ITyEnv} \times \text{IVarEnv} \end{array}$$

The typing rules are given in Figure 4. They allow inferences among sentences of the forms $\Gamma \vdash e : \tau$ and $\Gamma \vdash d : \Gamma'$, 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 and names Γ to denote the set of type names and 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.

$\boxed{\Gamma \vdash e : \tau \;\text{ and }\; \Gamma \vdash d : \Gamma'}$

$$\frac{\text{Dom}(\Gamma(t)) = \{c\} \qquad \Gamma(t)(c_i) \succ \tau^{(k)}t \qquad \Gamma \vdash e : \tau'}{\Gamma \vdash \lambda c : \tau^{(k)}t.e : \tau^{(k)}t \to \tau'} \quad (33)$$

$$\frac{\Gamma + \{x \mapsto \tau\} \vdash e : \tau' \qquad x \notin \text{names}\,\Gamma}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'} \quad (34)$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\; e_2 : \tau} \quad (35) \qquad \frac{\Gamma(x) \succ \tau}{\Gamma \vdash x_\tau : \tau} \quad (36)$$

$$\frac{\tau = \tau^{(k)}t \qquad \Gamma(t)(c) \succ \tau}{\Gamma \vdash c_\tau : \tau} \quad (37) \qquad \frac{\Gamma \vdash d : \Gamma' \qquad \Gamma + \Gamma' \vdash e : \tau \qquad \text{Dom}\,\Gamma' \cap \text{tynames}\,\tau = \emptyset}{\Gamma \vdash \texttt{let}\; d\; \texttt{in}\; e : \tau} \quad (38)$$

$$\frac{x \notin \text{names}\,\Gamma \qquad \Gamma \vdash e : \tau \qquad \{\alpha^{(k)}\} \cap \text{tyvars}\,\Gamma = \emptyset}{\Gamma \vdash \texttt{valdec}\; x : \alpha^{(k)}.\tau \;\texttt{=}\; e : \{x \mapsto \forall\alpha^{(k)}.\tau\}} \quad (39)$$

$$\frac{t \notin \text{names}\,\Gamma \qquad \text{arity}\; t = k}{\Gamma \vdash \texttt{datdec}\; \alpha^{(k)}t \;\texttt{=}\; c : \{t \mapsto \{c \mapsto \forall\alpha^{(k)}.\alpha^{(k)}t\}\}} \quad (40)$$

$$\frac{\Gamma \vdash d_1 : \Gamma_1 \qquad \Gamma + \Gamma_1 \vdash d_2 : \Gamma_2}{\Gamma \vdash d_1 \;;\; d_2 : \Gamma_1 + \Gamma_2} \quad (41) \qquad \frac{}{\Gamma \vdash \varepsilon : \{\}} \quad (42)$$

Figure 4: Typing rules for the intermediate language.

The requirement $\text{Dom}(\Gamma(t)) = \{c\}$ in rule 33 ensures that the simple pattern-matching mechanism is exhaustive. The side condition in rule 38 ensures that locally declared type names do not escape.

3.2 Dynamic Semantics

Evaluation of an intermediate language 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.

We also use e and d to range over untyped phrases; it is always clear from the context whether we talk about typed or untyped phrases. The erasure function *er* is defined by the equations

$$\begin{array}{rcl} er(\lambda a:\tau.e) & = & \lambda a.er(e) \\ er(e_1\; e_2) & = & er(e_1)\; er(e_2) \\ er(a_\tau) & = & a \\ er(\texttt{let}\; d\; \texttt{in}\; e) & = & \texttt{let}\; er(d)\; \texttt{in}\; er(e) \\ er(\texttt{valdec}\; x : \alpha^{(k)}.\tau \;\texttt{=}\; e) & = & \texttt{valdec}\; x \;\texttt{=}\; er(e) \\ er(\texttt{datdec}\; \alpha^{(k)}t \;\texttt{=}\; c) & = & \varepsilon \\ er(d_1 \;;\; d_2) & = & er(d_1) \;;\; er(d_2) \\ er(\varepsilon) & = & \varepsilon \end{array}$$

Notice that untyped declarations do not include datatype declarations.

The semantics of untyped intermediate language phrases is given as a natural operational semantics. The rules are instrumented with extra rules for expressing that evaluation

$\boxed{\mathcal{D} \vdash e \leadsto r \text{ and } \mathcal{D} \vdash d \leadsto \mathcal{D}'}$

$$\frac{}{\mathcal{D} \vdash \lambda a.e \leadsto \langle \lambda a.e, \mathcal{D} \rangle} \ (43) \qquad \frac{}{\mathcal{D} \vdash c \leadsto c} \ (44)$$

$$\frac{\mathcal{D}(x) = v}{\mathcal{D} \vdash x \leadsto v} \ (45) \qquad \frac{x \notin \text{Dom } \mathcal{D}}{\mathcal{D} \vdash x \leadsto \mathbf{wr}} \ (46)$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash e_1 \leadsto \langle \lambda x.e, \mathcal{D}_0 \rangle \\ \mathcal{D} \vdash e_2 \leadsto v \quad \mathcal{D}_0 + \{x \mapsto v\} \vdash e \leadsto r\end{array}}{\mathcal{D} \vdash e_1 \ e_2 \leadsto r} \ (47)$$

$$\frac{\mathcal{D} \vdash e_1 \leadsto \langle \lambda c.e, \mathcal{D}_0 \rangle \quad \mathcal{D} \vdash e_2 \leadsto c \quad \mathcal{D}_0 \vdash e \leadsto r}{\mathcal{D} \vdash e_1 \ e_2 \leadsto r} \ (48)$$

$$\frac{\mathcal{D} \vdash e_1 \leadsto \langle \lambda c.e, \mathcal{D}_0 \rangle \quad \mathcal{D} \vdash e_2 \leadsto r \quad c \neq r}{\mathcal{D} \vdash e_1 \ e_2 \leadsto \mathbf{wr}} \ (49)$$

$$\frac{\mathcal{D} \vdash e_1 \leadsto \mathbf{wr} \text{ or } \mathbf{c}}{\mathcal{D} \vdash e_1 \ e_2 \leadsto \mathbf{wr}} \ (50) \qquad \frac{\begin{array}{c}\mathcal{D} \vdash e_1 \leadsto \langle \lambda x.e, \mathcal{D}_0 \rangle \\ \mathcal{D} \vdash e_2 \leadsto \mathbf{wr}\end{array}}{\mathcal{D} \vdash e_1 \ e_2 \leadsto \mathbf{wr}} \ (51)$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash d \leadsto \mathcal{D}' \\ \mathcal{D} + \mathcal{D}' \vdash e \leadsto r\end{array}}{\mathcal{D} \vdash \texttt{let } d \texttt{ in } e \leadsto r} \ (52) \qquad \frac{\mathcal{D} \vdash d \leadsto \mathbf{wr}}{\mathcal{D} \vdash \texttt{let } d \texttt{ in } e \leadsto \mathbf{wr}} \ (53)$$

$$\frac{\mathcal{D} \vdash e \leadsto v}{\mathcal{D} \vdash \texttt{valdec } x \texttt{ = } e \leadsto \{x \mapsto v\}} \ (54)$$

$$\frac{\mathcal{D} \vdash e \leadsto \mathbf{wr}}{\mathcal{D} \vdash \texttt{valdec } x \texttt{ = } e \leadsto \mathbf{wr}} \ (55)$$

$$\frac{\begin{array}{c}\mathcal{D} \vdash d_1 \leadsto \mathcal{D}_1 \\ \mathcal{D} + \mathcal{D}_1 \vdash d_2 \leadsto \mathcal{D}_2\end{array}}{\mathcal{D} \vdash d_1 \ ; \ d_2 \leadsto \mathcal{D}_1 + \mathcal{D}_2} \ (56) \qquad \frac{\begin{array}{c}\mathcal{D} \vdash d_1 \leadsto \mathcal{D}_1 \\ \mathcal{D} + \mathcal{D}_1 \vdash d_2 \leadsto \mathbf{wr}\end{array}}{\mathcal{D} \vdash d_1 \ ; \ d_2 \leadsto \mathbf{wr}} \ (57)$$

$$\frac{\mathcal{D} \vdash d_1 \leadsto \mathbf{wr}}{\mathcal{D} \vdash d_1 \ ; \ d_2 \leadsto \mathbf{wr}} \ (58) \qquad \frac{}{\mathcal{D} \vdash \varepsilon \leadsto \{\}} \ (59)$$

Figure 5: Semantics of the intermediate language.

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 untyped intermediate language expressions. The semantic objects for the dynamic semantics are given as follows:

$$\begin{array}{rcl} v & \in & \text{Val} = \text{ICon} \cup \text{Clos} \\ \langle \lambda a.e, \mathcal{D} \rangle & \in & \text{Clos} = \text{IExp} \times \text{DynEnv} \\ \mathcal{D} & \in & \text{DynEnv} = \text{IVar} \xrightarrow{\text{fin}} \text{Val} \\ r & \in & \text{ExpResult} = \text{Val} \cup \{\mathbf{wr}\} \\ \varrho & \in & \text{DecResult} = \text{DynEnv} \cup \{\mathbf{wr}\} \end{array}$$

Constructors are values. The object **wr** 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 are given in Figure 5. The rules allow inferences among sentences of the forms $\mathcal{D} \vdash e \leadsto r$ and $\mathcal{D} \vdash d \leadsto \varrho$, where $\mathcal{D}$ is a dynamic environment, e is an untyped expression, d is an 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 $\mathcal{D}$."

In rules 48 and 49, the pattern-matching mechanism fails if the constructor in the closure is not identical to the result of evaluating the argument of the application.

Type soundness of the intermediate language states that well-typed intermediate language phrases cannot go wrong:

Proposition 3.1 (Type soundness) *If* $\{\} \vdash d : \Gamma$ *and* $\{\} \vdash er(d) \leadsto \varrho$ *then* $\varrho \neq \mathbf{wr}$.

PROOF The proof is an inductive argument on the structure of intermediate language phrases. The proof sets up a consistency relation between dynamic environments and type environments and uses the properties that the consistency relation and the typing rules are closed under substitution. Chapter 7 of [3] gives the full proof. □

The proof is inspired by other proofs of type soundness [14, 8] for Milner's polymorphic type discipline [10, 2]. The technique that we use to demonstrate soundness for the intermediate language extends to other features of Standard ML including datatypes with multiple value constructors [3] and recursion and imperative language constructs [8].

4 Interpretation of Modules

There are three important aspects to the interpretation of Modules. First, structures are flattened during interpretation. The interpretation avoids name clashes due to flattening by maintaining a mapping from source language identifiers to freshly generated intermediate language variables.

The second important aspect to the interpretation is that functors are specialised for each application. Although there is a potential possibility for an exponential increase in code size, Standard ML functors cannot be recursive, thus, the interpretation terminates. Moreover, experience with large software projects that use Standard ML Modules extensively, such as the Kit and the Standard ML of New Jersey compiler, indicates that few functors are applied more than once. We shall return to this issue in Section 6.

The third important aspect to the interpretation is that signature matching results in no intermediate language phrases; in the case a value component of a structure is made less polymorphic, 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. Consider the 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 intermediate language declaration

$$\begin{array}{l} \texttt{datdec } t \texttt{ = } c \texttt{ ;} \\ \texttt{valdec } x : \forall\alpha.\alpha \to \alpha \texttt{ = } \lambda y : \alpha.y_\alpha \texttt{ ;} \\ \texttt{valdec } z : t \texttt{ = } x_{t \to t} \ c \end{array}$$

where c is a constructor associated with the identifier `A`, t is a type name with arity 0, and x, y, and z are intermediate language variables associated with the identifiers `id`, `b`, and `a`, respectively.

In the following sections, we formalise the static interpretation of Modules.

4.1 Semantic Objects for Interpretation

The following additional semantic objects are used for interpretation:

$$\begin{array}{rcl}
\mathcal{SE} & \in & \text{TStrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{TEnv} \\
\mathcal{VE} & \in & \text{TValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{TValEntry} \\
(\sigma, is, a:\sigma') & \in & \text{TValEntry} = \text{TypeSch} \times \text{IdStatus} \times \\
 & & \quad (\text{IVar} \cup \text{ICon}) \times \text{TypeSch} \\
\mathcal{E} & \in & \text{TEnv} = \text{TStrEnv} \times \text{TyEnv} \times \text{TValEnv} \\
\mathcal{F} & \in & \text{TFunEnv} = \text{FunId} \xrightarrow{\text{fin}} \\
 & & \quad (\text{TBasis} \times \text{StrId} \times \text{StrExp} \times \text{FunSig}) \\
\mathcal{B} & \in & \text{TBasis} = \text{TFunEnv} \times \text{TEnv}
\end{array}$$

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 its identifier status (v or c), and $a:\sigma'$ is a pair of an intermediate language variable and its type scheme. Because a value component of a structure can be made less polymorphic by signature matching, the type scheme σ' can be more general than the type scheme σ.

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 of the functor.

4.2 Weakening and Enlargement

When $\mathcal{A}$ is some translation environment or translation basis, we define the *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:

$$\begin{array}{rcl}
\overline{(\mathcal{F},\mathcal{E})} & = & (\overline{\mathcal{F}},\overline{\mathcal{E}}) \\
\overline{(\mathcal{B}, strid, strexp, \Phi)} & = & \Phi \\
\overline{(\mathcal{SE}, TE, \mathcal{VE})} & = & (\overline{\mathcal{SE}}, TE, \overline{\mathcal{VE}}) \\
\overline{(\sigma, is, a:\sigma')} & = & (\sigma, is) \\
\overline{\mathcal{SE}} & = & \{strid \mapsto \overline{\mathcal{SE}(strid)} \mid strid \in \text{Dom}\ \mathcal{SE}\} \\
\overline{\mathcal{VE}} & = & \{vid \mapsto \overline{\mathcal{VE}(vid)} \mid vid \in \text{Dom}\ \mathcal{VE}\} \\
\overline{\mathcal{F}} & = & \{funid \mapsto \overline{\mathcal{F}(funid)} \mid funid \in \text{Dom}\ \mathcal{F}\}
\end{array}$$

Enlargement relates translation environments much as enrichment relates elaboration environments. An object $(\sigma_1, is_1, a_1:\sigma_1')$ *enlarges* an 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 $\mathcal{VE}_2$

4.3 From Core to the Intermediate Language

We first present rules for translating Core phrases into intermediate language 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 Core expression, τ is a type, and e is an intermediate language 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 Core declaration, N is a set of names, and d is an intermediate language 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 $\mathcal{E}$." In objects of the form $(N)(\mathcal{A}, d)$, where $\mathcal{A}$ is either a translation environment or a translation basis, the prefix (N) 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 $(\mathcal{A}, d)$. The rules are given in Figure 6.

4.4 Interpretation of Modules

The rules for interpreting Modules phrases into intermediate language 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 intermediate language declaration; sentences of this form are read "*phrase* translates to $(N)(\mathcal{A}, d)$ in $\mathcal{B}$." The rules are given in Figure 7. Notice that no code is generated for structure bindings (rules 70 and 73). In rule 78, the interpretation of the functor body is delayed until the functor is applied.

The following proposition states that well-typed source language programs are translatable:

Proposition 4.1 (Translatability) *If* $\{\} \vdash topdec \Rightarrow (T)B$ *then there exists* $(N)(\mathcal{B}, d)$ *such that* $\{\} \vdash topdec \Rightarrow (N)(\mathcal{B}, d)$ *and* $N \supseteq T$ *and* $\overline{\mathcal{B}} = B$.

PROOF The proof is based on Proposition 2.1 and on the relationship between enrichment and enlargement [3, Chapter 8]. □

To state a type correctness property for the interpretation, we first define a consistency relation $\Gamma \models_{tc} \mathcal{B}$ between intermediate language type environments and translation bases. Type consistency expresses that all intermediate language variables and constructors in value entries in $\mathcal{B}$ are associated to the same type schemes as in Γ. The relation is defined inductively by the following equations:

- $\Gamma \models_{tc} (\mathcal{SE}, TE, \mathcal{VE})$ iff $\Gamma \models_{tc} \mathcal{E}$ for all $\mathcal{E} \in$ Ran $\mathcal{SE}$ and $\Gamma(x) = \sigma'$ for all $(\sigma, is, x:\sigma') \in$ Ran $\mathcal{VE}$ and $\Gamma(t)(c) = \sigma'$ and $\sigma' = \forall\alpha^{(k)}.\alpha^{(k)}t$ for all $(\sigma, is, c:\sigma') \in$ Ran $\mathcal{VE}$
- $\Gamma \models_{tc} (\mathcal{F}, \mathcal{E})$ iff $\Gamma \models_{tc} \mathcal{E}$ and $\Gamma \models_{tc} \mathcal{B}$ for all $(\mathcal{B}, strid, strexp, \Phi) \in$ Ran $\mathcal{F}$

Type correctness is then stated as follows:

Proposition 4.2 (Type correctness) *If* $\{\} \vdash topdec \Rightarrow (N)(\mathcal{B}, d)$ *then there exists* Γ *such that* $\{\} \vdash d : \Gamma$ *and* $\Gamma \models_{tc} \mathcal{B}$ *and* Dom $\Gamma \subseteq N$.

PROOF The proof uses an inductive argument on the structure of source language constructs [3, Chapter 8]. □

$\boxed{\mathcal{E} \vdash exp \Rightarrow \tau, e \;\text{ and }\; \mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d)}$

$$\frac{\mathcal{E}(longvid) = (\sigma, is, a : \sigma') \qquad \sigma' \succ \tau}{\mathcal{E} \vdash longvid \Rightarrow \tau, a_\tau} \quad (60)$$

$$\frac{\begin{array}{c} vid \notin \mathrm{Dom}\,\mathcal{E} \;\text{ or }\; is \text{ of } \mathcal{E}(vid) = \mathtt{v} \\ x \notin \mathrm{names}\,\mathcal{E} \qquad \mathcal{E} + \{vid \mapsto (\tau, \mathtt{v}, x : \tau)\} \vdash exp \Rightarrow \tau', e \end{array}}{\mathcal{E} \vdash \mathtt{fn}\; vid \;\mathtt{=>}\; exp \Rightarrow \tau \to \tau', \lambda x : \tau.e} \quad (61)$$

$$\frac{\mathcal{E}(longvid) = (\sigma, \mathtt{c}, c : \sigma) \qquad \sigma \succ \tau \qquad \mathcal{E} \vdash exp \Rightarrow \tau', e}{\mathcal{E} \vdash \mathtt{fn}\; longvid \;\mathtt{=>}\; exp \Rightarrow \tau \to \tau', \lambda c : \tau.e} \quad (62)$$

$$\frac{\mathcal{E} \vdash exp_1 \Rightarrow \tau' \to \tau, e_1 \qquad \mathcal{E} \vdash exp \Rightarrow \tau', e_2}{\mathcal{E} \vdash exp_1 \; exp_2 \Rightarrow \tau, e_1 \; e_2} \quad (63)$$

$$\frac{\begin{array}{c} \mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}', d) \\ N \cap \mathrm{names}(\mathcal{E}, \tau) = \emptyset \qquad \mathcal{E} + \mathcal{E}' \vdash exp \Rightarrow \tau, e \end{array}}{\mathcal{E} \vdash \mathtt{let}\; dec \;\mathtt{in}\; exp \;\mathtt{end} \Rightarrow \tau, \mathtt{let}\; d \;\mathtt{in}\; e} \quad (64)$$

$$\frac{\begin{array}{c} \{\alpha^{(k)}\} \cap \mathrm{tyvars}\,\mathcal{E} = \emptyset \qquad \sigma = \forall \alpha^{(k)}.\tau \\ \mathcal{E} \vdash exp \Rightarrow \tau, e \qquad \mathcal{E}' = \{vid \mapsto (\sigma, \mathtt{v}, x : \sigma)\} \qquad x \notin \mathrm{names}\,\mathcal{E} \end{array}}{\mathcal{E} \vdash \mathtt{val}\; vid \;\mathtt{=}\; exp \Rightarrow (\{x\})(\mathcal{E}', \mathtt{valdec}\; x : \alpha^{(k)}.\tau \;\mathtt{=}\; e)} \quad (65)$$

$$\frac{\mathrm{arity}\; t = k \qquad \sigma = \forall \alpha^{(k)}.\alpha^{(k)} t \qquad VE = \{vid \mapsto (\sigma, \mathtt{c})\} \qquad \mathcal{VE} = \{vid \mapsto (\sigma, \mathtt{c}, c : \sigma)\}}{\mathcal{E} \vdash \mathtt{datatype}\; \alpha^{(k)}\; tycon \;\mathtt{=}\; vid \Rightarrow (\{t, c\})((\{tycon \mapsto (t, VE)\}, \mathcal{VE}), \mathtt{datdec}\; \alpha^{(k)} t \;\mathtt{=}\; c)} \quad (66)$$

$$\frac{\overline{\mathcal{E}} \vdash ty \Rightarrow \tau \qquad \mathcal{E}' = \{tycon \mapsto (\Lambda \alpha^{(k)}.\tau, \{\})\}}{\mathcal{E} \vdash \mathtt{type}\; \alpha^{(k)}\; tycon \;\mathtt{=}\; ty \Rightarrow (\emptyset)(\mathcal{E}', \varepsilon)} \quad (67)$$

$$\frac{\mathcal{E}(longstrid) = \mathcal{E}'}{\mathcal{E} \vdash \mathtt{open}\; longstrid \Rightarrow (\emptyset)(\mathcal{E}', \varepsilon)} \quad (68)$$

Figure 6: Translation of Core.

$\boxed{\mathcal{B} \vdash strexp \Rightarrow (N)(\mathcal{E}, d) \;\text{ and }\; \mathcal{B} \vdash strdec \Rightarrow (N)(\mathcal{E}, d) \;\text{ and }\; \mathcal{B} \vdash topdec \Rightarrow (N)(\mathcal{B}', d)}$

$$\frac{\mathcal{E} \vdash dec \Rightarrow (N)(\mathcal{E}, d)}{(\mathcal{F}, \mathcal{E}) \vdash dec \Rightarrow (N)(\mathcal{E}, d)} \quad (69)$$

$$\frac{\mathcal{B} \vdash strexp \Rightarrow (N)(\mathcal{E}, d)}{\mathcal{B} \vdash \mathtt{structure}\; strid \;\mathtt{=}\; strexp \Rightarrow (N)(\{strid \mapsto \mathcal{E}\}, d)} \quad (70)$$

$$\frac{\begin{array}{c} \mathcal{B} \vdash strdec_1 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \qquad (N_1 \cup N_2) \cap \mathrm{names}\,\mathcal{B} = \emptyset \\ \mathcal{B} + \mathcal{E}_1 \vdash strdec_2 \Rightarrow (N_2)(\mathcal{E}_2, d_2) \qquad N_2 \cap (N_1 \cup \mathrm{names}(\mathcal{E}_1, d_1)) = \emptyset \end{array}}{\mathcal{B} \vdash strdec_1 \; strdec_2 \Rightarrow (N_1 \cup N_2)(\mathcal{E}_1 + \mathcal{E}_2, d_1 \;;\; d_2)} \quad (71)$$

$$\frac{}{\mathcal{B} \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \varepsilon)} \quad (72)$$

$$\frac{\mathcal{B} \vdash strdec \Rightarrow (N)(\mathcal{E}, d)}{\mathcal{B} \vdash \mathtt{struct}\; strdec \;\mathtt{end} \Rightarrow (N)(\mathcal{E}, d)} \quad (73)$$

$$\frac{\mathcal{B}(longstrid) = \mathcal{E}}{\mathcal{B} \vdash longstrid \Rightarrow (\emptyset)(\mathcal{E}, \varepsilon)} \quad (74)$$

$$\frac{\mathcal{B} \vdash strexp \Rightarrow (N)(\mathcal{E}, d) \qquad \overline{\mathcal{B}} \vdash sigexp \Rightarrow \Sigma \qquad \Sigma \geq \overline{\mathcal{E}'} \qquad \mathcal{E} \gg \mathcal{E}' \qquad N \cap \mathrm{names}\,\mathcal{B} = \emptyset}{\mathcal{B} \vdash strexp \;\mathtt{:}\; sigexp \Rightarrow (N)(\mathcal{E}', d)} \quad (75)$$

$$\frac{\begin{array}{c} \mathcal{B} \vdash strexp \Rightarrow (N)(\mathcal{E}, d) \qquad \mathcal{B}(funid) = (\mathcal{B}_0, strid, strexp_0, \Phi) \qquad \Phi \geq (\overline{\mathcal{E}'}, (T')\overline{\mathcal{E}_1}) \qquad T' \subseteq N_1 \qquad \mathcal{E} \gg \mathcal{E}' \\ (N \cup N_1) \cap \mathrm{names}\,\mathcal{B} = \emptyset \qquad N_1 \cap (N \cup \mathrm{names}(\mathcal{E}, d)) = \emptyset \qquad \mathcal{B}_0 + \{strid \mapsto \mathcal{E}'\} \vdash strexp_0 \Rightarrow (N_1)(\mathcal{E}_1, d_1) \end{array}}{\mathcal{B} \vdash funid \;\mathtt{(}\; strexp \;\mathtt{)} \Rightarrow (N \cup N_1)(\mathcal{E}_1, d \;;\; d_1)} \quad (76)$$

$$\frac{\mathcal{B} \vdash strdec \Rightarrow (N)(\mathcal{E}, d)}{\mathcal{B} \vdash strdec \Rightarrow (N)((\{\}, \mathcal{E}), d)} \quad (77)$$

$$\frac{\overline{\mathcal{B}} \vdash sigexp \Rightarrow (T)E \qquad T \cap \mathrm{tynames}\,\mathcal{B} = \emptyset \qquad \overline{\mathcal{B}} + \{strid \mapsto E\} \vdash strexp \Rightarrow \Sigma}{\begin{array}{c} \mathcal{B} \vdash \mathtt{functor}\; funid \;\mathtt{(}\; strid \;\mathtt{:}\; sigexp \;\mathtt{)}\; \mathtt{=}\; strexp \Rightarrow \\ (\emptyset)(\{funid \mapsto (\mathcal{B}, strid, strexp, (T)(E, \Sigma))\}, \varepsilon) \end{array}} \quad (78)$$

$$\frac{\begin{array}{c} \mathcal{B} \vdash topdec_1 \Rightarrow (N_1)(\mathcal{B}_1, d_1) \qquad (N_1 \cup N_2) \cap \mathrm{names}\,\mathcal{B} = \emptyset \\ \mathcal{B} + \mathcal{B}_1 \vdash topdec_2 \Rightarrow (N_2)(\mathcal{B}_2, d_2) \qquad N_2 \cap (N_1 \cup \mathrm{names}(\mathcal{B}_1, d_1)) = \emptyset \end{array}}{\mathcal{B} \vdash topdec_1 \; topdec_2 \Rightarrow (N_1 \cup N_2)(\mathcal{B}_1 + \mathcal{B}_2, d_1 \;;\; d_2)} \quad (79)$$

$$\frac{}{\mathcal{B} \vdash \varepsilon \Rightarrow (\emptyset)(\{\}, \varepsilon)} \quad (80)$$

Figure 7: Interpretation of Modules.

5 From Opaque to Transparent Modules

In Standard ML, *opaque signature matching*, that is, structure expressions on the form *strexp* `:>` *sigexp*, makes it possible to hide implementation details of a type declared in *strexp* to the degree to which the type is specified in *sigexp*. In this section, we shall see that it is possible to translate opaque signature matching into transparent signature matching (i.e., structure expressions on the form *strexp* `:` *sigexp*) in such a way that elaboration is preserved under related assumptions. This translation, which is called *opacity elimination*, is important because it allows static interpretation to expose implementation details of abstract types into the intermediate language (which is of importance to what analyses are possible on the intermediate language.) Opacity elimination is defined as a function *oe* from program phrases to program phrases. Here are a couple of the defining equations:

$$\begin{array}{rcl} oe(strexp \ \texttt{:>}\ sigexp) & = & strexp \ \texttt{:}\ sigexp \\ oe(funid\ \texttt{(}\ strexp\ \texttt{)}) & = & funid\ \texttt{(}\ oe(strexp)\ \texttt{)} \\ & \vdots & \end{array}$$

Although opacity elimination is a straightforward translation, it is not trivial to show that it preserves elaboration. To do so, we first extend the typing rules of Figure 3 with a rule for opaque signature matching:

$$\frac{B \vdash strexp \Rightarrow (T)E \qquad B \vdash sigexp \Rightarrow \Sigma \qquad \Sigma \geq E' \prec E \qquad T \cap \text{tynames}\, B = \emptyset}{B \vdash strexp\ \texttt{:>}\ sigexp \Rightarrow \Sigma} \tag{81}$$

This rule differs from the rule for transparent signature matching (i.e., rule 23) in that the result of the matching is exactly the signature obtained by typing the signature expression. The addition of opaque signature matching to the language does not violate any of the previous properties that we have shown.

We now give three examples, so as to justify how the typings of a program phrase and its translated phrase relate by a notion of abstraction. Consider the functor declaration

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

and let B_0 be the basis $\{\texttt{int} \mapsto (t_{\text{int}}, \{\})\}$, 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`:

$$(\emptyset)(\{\}, (\emptyset)\{\texttt{a} \mapsto (t_{\text{int}}, \{\})\})$$

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 the second example, consider the functor declaration

```
functor G(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

$$(\{t\})(\{\texttt{a} \mapsto (t, \{\})\}, (\{t'\})\{\texttt{b} \mapsto (t', \{\})\})$$

for `G`, where t and t' are type names with arity 0. The functor declaration translates into the functor declaration

```
functor G(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

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

where t is a type name with arity 0. Also in this case, opacity elimination decreases generativity—or type abstraction.

It is not always the case, however, that generativity is decreased by opacity elimination. In fact, in this final example, we shall see that opacity elimination may increase generativity. Consider the functor declaration

```
functor H() = struct datatype a = A
                     datatype 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

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

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

```
functor H() = struct datatype a = A
                     datatype 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

$$(\emptyset)(\{\}, (\{t, t'\})\{\texttt{c} \mapsto (\Lambda().t \rightarrow t', \{\})\})$$

where t and t' are type names with arity 0.

We now turn to a formal definition of the abstraction relation.

5.1 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 $\Sigma_1 \cap T_2 = \emptyset$. One can show that signature abstraction is closed under realisation, that is, if $\Sigma_1 \succeq \Sigma_2$ then $\varphi(\Sigma_1) \succeq \varphi(\Sigma_2)$ for any realisation φ.

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 $\text{Dom}\ F_1 = \text{Dom}\ F_2$ and $F_1(funid) \succeq F_2(funid)$, for all $funid \in \text{Dom}\ F_1$. 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) $\text{Supp}\ \varphi \subseteq T_1$, (2) $\varphi(B_1) \succeq B_2$, and (3) $T_2 \cap \text{tynames}((T_1)B_1) = \emptyset$.

5.2 Preservation of Elaboration

We can now state the proposition saying that opacity elimination preserves typeability under related assumptions:

Proposition 5.1 (Preservation of elaboration) *Let phrase be either a structure-level expression, a structure-level declaration, or a top-level declaration. If* $\{\} \vdash$ *phrase* $\Rightarrow A$ *then there exists a semantic object* A' *such that* $\varphi(A) \succeq A'$ *and* $\{\} \vdash oe(phrase) \Rightarrow A'$.

Proof The proof is by induction over the structure of *phrase* and depends on another proposition saying that typeability of signature expressions is closed under abstraction and realisation. Details are given in [3, Chapter 5]. □

An interesting aspect of the proof that opacity elimination preserves typeability 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. Moreover, the translation fits into the framework of smart recompilation.

6 The ML Kit with Regions

The ML Kit with Regions (the Kit) is a Standard ML compiler based on region inference [16]. The Kit implements Standard ML Modules using static interpretation and the framework for smart recompilation that we discussed in the introduction.

The 33,000 lines Standard ML program AnnoDomini, which is a tool for correcting year 2000 problems in OSVS COBOL programs, has been compiled with the Kit. A region profile of running AnnoDomini (which include parsing, type checking, and program transformation) on a 1500 lines OSVS COBOL program is shown in Figure 8. Notice that type checking of the 1500 line program is performed in constant space. To obtain this memory behavior, it is essential that region-annotated types are propagated across module boundaries.

The Kit has also successfully compiled itself—about 87,000 lines of Standard ML. In doing so, only six of the 138 functors that the Kit consists of were compiled more than once. Although two of these functors, which implement sets and maps based on an ordering relation, were compiled 9 and 11 times, respectively, these functors are small. Based on the sizes of the source files, a simple calculation shows that for compiling the Kit, static interpretation increases the number of compiled lines by only 10.9 percent. Here we have not accounted for the fact that static interpretation decreases code size by eliminating all modules language constructs at compile time.

7 Related Work

The idea of eliminating parameterised modules at compile time is not new. C++ templates and Ada packages are usually implemented by expanding the body of the parameterised module at the points where the module is used. Moreover, tools are available for eliminating functors from Standard ML programs [4, 19], but these tools do not support any form of separate compilation and have not been formally justified.

The static semantics of our source language uses type abstraction rather than unique stamps [11] to model type generativity. Type abstraction is used in other type systems for module languages to model abstract types and type dependencies [5, 9]. Independently from the work we have done here, Russo [12] has developed a static semantics for a subset of Standard ML Modules, much similar to the static semantics for the source language that we present here. In particular, Russo also formulates type generativity by type abstraction. Moreover, he demonstrates that his static semantics accepts the same set of language phrases as a name based static semantics. Russo does not demonstrate type soundness for his language.

Similar to the approach we take here, Harper and Stone [6] 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, static interpretation eliminates all Modules language constructs during interpretation.

Shao has proposed a framework for cross module optimisation that allows information to propagate across module boundaries [13]. However, certain advanced analyses such as region inference (which do not immediately work for the FLINT variant of the F^{ω} language) do not carry over to module languages in his framework.

8 Conclusion

Static interpretation is a novel approach to the compilation of Standard ML Modules. The technique has several advantages. First, it allows information about identifiers to propagate across module boundaries. Second, the technique makes available (for the compiler) the exact definition of abstract types. Finally, the technique introduces no overhead for programming with modules, as no code is generated for Modules language constructs. The first two properties are essential to make advanced analyses such as region inference applicable for the entire Standard ML language.

The technique is used in the ML Kit with Regions compiler [16] and combined with a framework for smart recompilation, which makes static interpretation useful for compiling large programs. The Kit has yet to exploit the possibilities of propagating other than region information across module boundaries. Other possibilities for compiler optimisation that static interpretation enables to work across module boundaries include in-lining of small functions and optimisation of data representations (e.g., unboxing of floating-point values and flattening of arrays and list elements.)

Acknowledgments

I would like to thank my PhD supervisor Mads Tofte for great advise about many aspects of the work presented here. I also want to thank Niels Hallenberg, Xavier Leroy, Greg Morrisett, Hanne Riis Nielson, Tommy Højfeld Olesen, and Peter Sestoft, who all contributed with valuable comments about this work.

References

[1] 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.

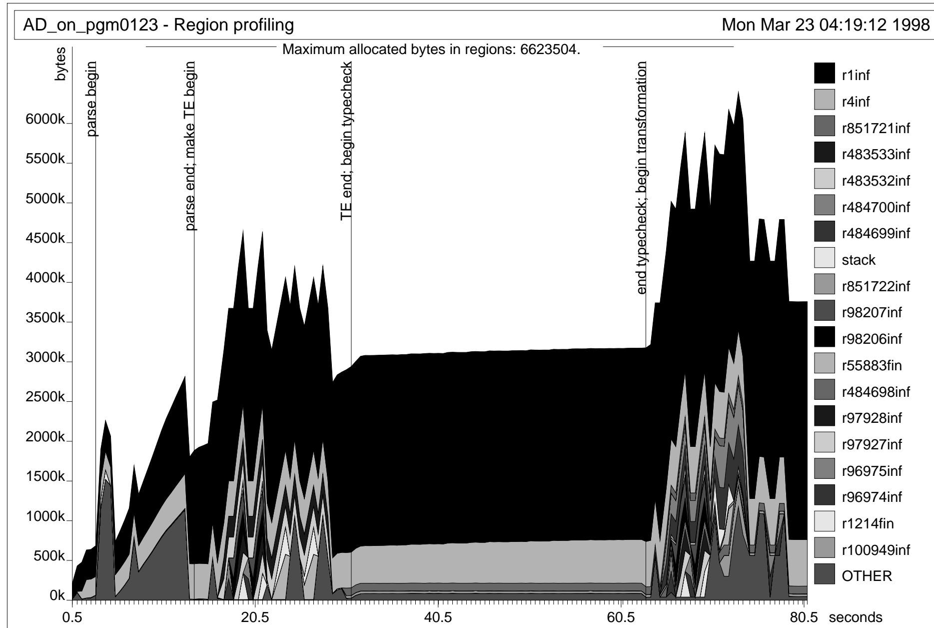

Figure 8: A region profile of running AnnoDomini on a 1500 lines OSVS COBOL program.

[2] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *9th ACM Symposium on Principles of Programming Languages*, January 1982.

[3] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation.* PhD thesis, Department of Computer Science, University of Copenhagen, January 1999.

[4] The Berkeley ANalysis Engine Group. The BANE de-functorizer. Available via the URL http://www.cs.berkeley.edu/Research/Aiken/bane.html.

[5] 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.

[6] Robert Harper and Chris Stone. An interpretation of Standard ML in type theory. Technical report, Carnegie Mellon University, June 1997. CMU-CS-97-147.

[7] 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.

[8] Xavier Leroy. *Polymorphic Typing of an Algorithmic Language.* PhD thesis, INRIA, October 1992.

[9] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st ACM Symposium on Principles of Programming Languages*, pages 109–122, 1994.

[10] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[11] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[12] Claudio V. Russo. *Types for Modules.* PhD thesis, University of Edinburgh, Department of Computer Science, June 1998.

[13] Zhong Shao. Typed cross-module compilation. Technical report, Department of Computer Science, Yale University, July 1997. YALEU/DCS/TR-1126.

[14] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, Department of Computer Science, May 1988.

[15] Mads Tofte and Lars Birkedal. A region inference algorithm. *Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):734–767, July 1998.

[16] 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.

[17] 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.

[18] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[19] Stephen Weeks. `smlc` user's guide. A whole-program Standard ML compiler, August 1998.