Smartest Recompilation

Zhong Shao and Andrew W. Appel
Department of Computer Science, Princeton University
Princeton, NJ 08544-2087
zsh@cs.princeton.edu appel@cs.princeton.edu

Abstract

To separately compile a program module in traditional statically-typed languages, one has to manually write down an import interface which explicitly specifies all the external symbols referenced in the module. Whenever the definitions of these external symbols are changed, the module has to be recompiled. In this paper, we present an algorithm which can automatically infer the "minimum" import interface for any module in languages based on the Damas-Milner type discipline (e.g., ML). By "minimum", we mean that the interface specifies a set of assumptions (for external symbols) that are just enough to make the module type-check and compile. By compiling each module using its "minimum" import interface, we get a separate compilation method that can achieve the following optimal property: *A compilation unit never needs to be recompiled unless its own implementation changes.*

1 Introduction

Most traditional separate compilation methods rely on manually created contexts (e.g., Modula-3 interfaces, "include-files" in C, and Ada package specifications) to enforce type correctness across module boundaries. Using the proper contexts, the compiler can check that each module uses its imported interfaces properly, and implements its exported interface as expected. The disadvantage of using these manually created contexts is that to guarantee consistency, all modules using a changed context must be recompiled, no matter how small the change is. The conventional recompilation rule (as described in Tichy [31]) is stated as follows: *"A compilation unit must be recompiled whenever* (1) *its own implementation changes, or* (2) *a context changes upon which the compilation unit depends."* This is obviously not satisfactory because adding a comment or adding a new declaration to a pervasive context may cause the unnecessary recompilation of the entire system. Tichy [31] presents an effective technique called "smart recompilation" that eliminates most of the redundant recompilations triggered by (2). In Tichy's scheme, a compilation unit is recompiled only if its implementation changes, or if it references a symbol defined elsewhere whose definition has changed. Schwanke and Kaiser [29] define "smarter recompilation" which can eliminate even more (but not all) redundant recompilations caused by (2). So a natural question to ask is: Can we eliminate all redundant recompilations? that is, can we achieve the following **smartest recompilation rule**: *A compilation unit never needs to be recompiled unless its own implementation (source code) changes*?

Standard ML (SML) [20] has a rather elaborate module system, but SML compilers have not supported separate compilation very well. The problem is that in SML, modules such as structures and functors can liberally reference externally defined identifiers without even mentioning what are their specifications. For example, by using "qualified" (dotted) identifiers, a structure `FOO` can use `BAR.QUX.f` to reference the function `f` defined in the substructure `QUX` of the structure `BAR`, without even knowing what the type of `f` is. Because of the lack of explicit import interfaces, structures and functors with free variables (let's call them open-formed modules) are not considered as separately compilable units. So how can we separately compile open-formed modules in SML?

This paper presents a new separate compilation method which actually answers both of the above two questions. Surprisingly, not only can we separately compile arbitrary structures and functors in SML, but we can also accomplish the "smartest recompilation rule." Our idea is simple: in order to separately compile a module with references to external identifiers, we have to know the specifications (e.g., types) of these external identifiers; since they are not explicitly specified, we infer them by looking at how these external identifiers are used inside the module; then we compile the module by using this inferred import interface as its context; finally, when all the modules are linked together, cross-module type errors are reported by checking whether the surroundings match (or satisfy) the specifications in each inferred import interface. The catch here is that in order to achieve the "smartest recompilation rule", we have to infer the "minimum" import interface. Informally speaking, this "minimum" import interface specifies a set of assumptions (on those external identifiers) that are just enough to make the module type-check and compile; at link time, if the module's surroundings satisfy this set of assumptions, the compiled code can be reused, otherwise there must be cross-module type errors. The inference algorithm is discussed in detail in section 2 and 3.

ACM-20th PoPL-1/93-S.C., USA
© 1993 ACM 0-89791-561-5/93/0001/0439...$1.50

Now let's see an example of how our method works. From the following SML structure declaration,

```
structure FOO = struct
                  val x = BAR.f
                  val y = (BAR.g 4, BAR.g true)
                end
```

we know that in order to compile FOO, the context should contain a structure named BAR. Inside BAR, there should be at least two val declarations: one is f, which can have any type, say α; the other is g, which should be a function that can be applied to both integers and booleans, that is, g should have a type more general than $int \rightarrow \beta$ and $bool \rightarrow \gamma$. Here, α, β and γ are just type variables we used to denote unknown types. Compiling FOO in this inferred interface will result in a structure with two components: the variable x has type α and the variable y has type $\beta * \gamma$. Now suppose that the real structure BAR is defined as follows:

```
structure BAR = struct val f = 3
                       val h = true
                       val g = fn z => z
                end
```

then at link time, when the real BAR is matched against the import interface of FOO, we find that the type variables α and β should be *int* and γ should be *bool*, thus FOO.x will have type *int* and FOO.y will have type $int * bool$. This is exactly what we will get if we compile FOO in the environment that would result from compiling BAR.

To achieve the "smartest recompilation rule", the back end of the compiler must use only the type information specified in the module's inferred import interface. This limitation is not a big problem. The back end of the current SML/NJ compiler [4] uses almost no type information from the front end but it still produces quite efficient code. Many optimization techniques that do use the type information, such as Leroy's representation analysis [17], can still be partially incorporated into our separate compilation system. The details will be described in section 4 and 5 of this paper.

Our separate compilation method immediately has the following advantages over traditional methods:

- Because of the smartest recompilation rule, each module never needs to be recompiled unless its own implementation changes; so maximum reusability is achieved.
- Because all modules are compiled independently of each other, they can be compiled in any order. This also means that programmers need no longer maintain dependency files (e.g., Makefile [9]).
- Open-formed modules can also be separately compiled.
- Cross-module type errors are now symmetric. In traditional methods, if module A references identifiers defined in module B, type inconsistencies between module A and module B will show up when A is compiled. If the programmer fixes the error by editing B, both A and B must be recompiled. But in our method, because cross-module type errors are reported at link time, only B will be recompiled.
- The compiler based on our method will automatically be a standalone compiler. As far as we know, no one has yet built a standalone compiler for the complete SML module system.

1.1 Closed vs. open-formed modules

Standard ML allows programming in open-formed modules. The essential difference between closed and open-formed modules can be seen from rewriting the above open-formed structure FOO in closed form, the SML functor FOO'.

```
functor FOO' (BAR : sig val f : int
                        val g : 'a -> 'a
                    end)
  = struct val x = BAR.f
           val y = (BAR.g 4, BAR.g true)
    end
```

This shows that programmers have to give assumptions about the types of BAR.f and BAR.g based on a *pro forma* implementation of the structure declaration BAR. If later, BAR.g is changed to have a type scheme $\forall\alpha.\alpha \rightarrow int$, the above declaration will no longer be useful and will have to be modified and recompiled. An open-formed module such as structure FOO does not have this problem because it does not require that the specifications of imported identifiers be explicitly given. People may want to write FOO' using its minimum import interface as its argument signature, but this "minimum" is not expressible in the SML type system. Moreover, inferring the minimum import interface cannot be easily done by hand.

We do not advocate writing large programs all in open-formed modules. SML strongly encourages that every structure declaration should be written with a result signature constraint (as its export interface). Programmers can write their programs all in the form of closed functors such as FOO'. On the other hand, in practice, we find it extremely convenient and flexible to write parts of our programs as open-formed modules.

For languages based on the Damas-Milner type discipline [7] such as SML and Haskell [11], there is another reason in favor of writing certain modules in opened forms. One of the most important features of the Damas-Milner type discipline is that the most general type for arbitrary expressions can be automatically inferred by compilers. It is, however, nontrivial to infer the most general type simply by hand, especially with the presence of polymorphic references in SML or type classes in Haskell. This makes it also nontrivial to write explicit import interfaces for many modules. The SML commentary [19] also suggests that programmers will probably need to write down many sharing equations if they want to close every module and that it will be too restrictive to write everything in closed functors.

2 Assumption Inference in Core ML

ML has a sophisticated type inference system. Given an ML expression $e = \lambda x.f(x+1)$, even though the type of newly introduced variable x is not specified, we can still find the most general type of e if we know the type of f and $+$. For example, if f has type $\forall\alpha.\alpha \rightarrow \alpha$ and $+$ has type $int * int \rightarrow int$, e will have type $int \rightarrow int$. Milner's type inference (or type reconstruction) algorithm W (as in Tofte [32]) takes two arguments, a type environment TE and an ML expression e; all the free variables in e (such as f and $+$) must be specified with a type in TE, and $W(TE, e)$ will return the most general type for e.

To support "smartest recompilation," we face the challenge of doing type inference without even knowing the type

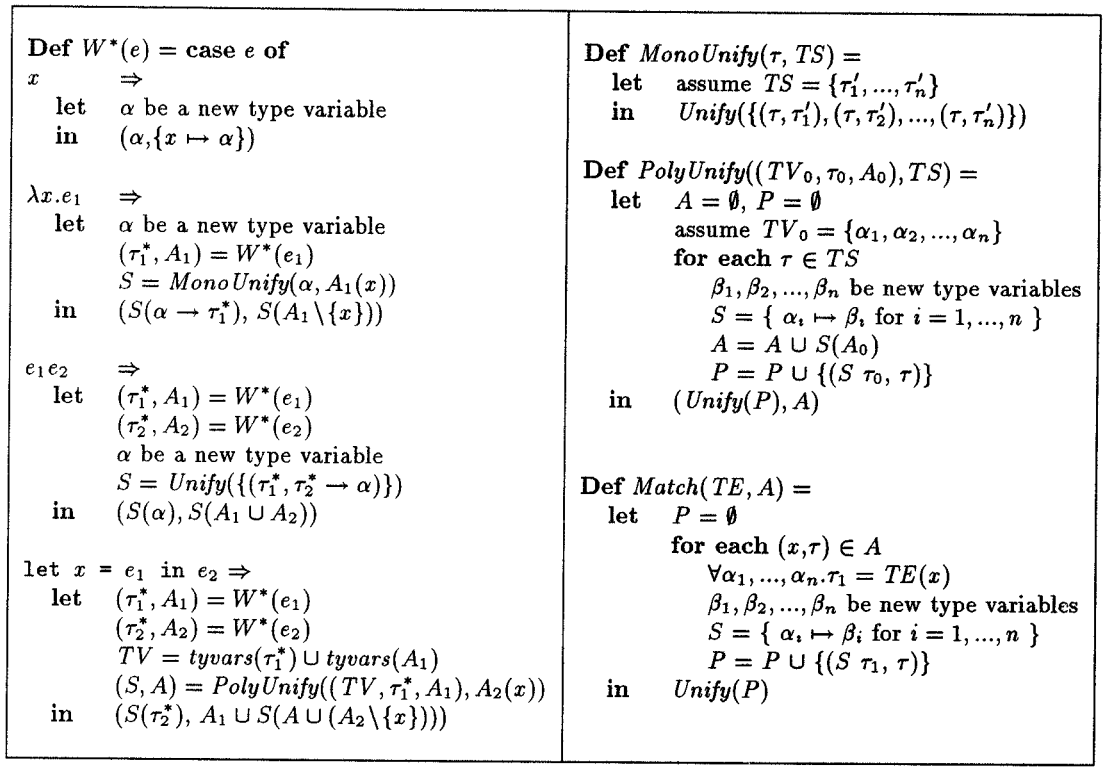

Figure 1: The Inference Algorithm W^* and *Match*

of external identifiers. For example, can we find out the most general type of the above expression e if we do not know the type of f and $+$? This seems impossible. But in the case of separate compilation, we assume that the types of external identifiers will be known at link time. We can divide the type inference into two phases: first (at compile time), we infer a type τ for e and a set of assumptions A for the free variables in e, which essentially means that e will have type τ if the free variables in e satisfy the constraints in A; then (at link time) when the types of those free variables (i.e., TE) are known, we match them against those in A and "magically" recover the most general type of e in TE. To distinguish it from usual ML type inference, we call the inference done in the first phase "assumption inference".

In this section and section 3 we discuss the details of our assumption inference algorithm and show how the matching done at link time can successfully recover the correct type for each expression. To simplify the presentations, we divide our algorithms into two parts: this section for Core ML and the next section for the ML module language. We only give the details of our algorithm for the mini-ML language *Exp* and the skeletal module language *ModL* used by Tofte [32]. However it is easy to extend our algorithm to the rest of SML.

The expressions in the mini-ML language *Exp* are defined by the following grammar:

$$e ::= x \mid \lambda x.e_1 \mid e_1\ e_2 \mid \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

Here is a brief review of the notation. Suppose TyVar is an infinite set of type variables and TyCon is a set of nullary type constructors, the set of *types*, Type, ranged over by τ and the set of *type schemes*, TypeScheme, ranged over by σ are defined by $\tau ::= \pi \mid \alpha \mid \tau_1 \to \tau_2$ and $\sigma ::= \tau \mid \forall \alpha.\sigma_1$. A *type environment* is a finite map from program variables to type schemes. $tyvars(\tau)$, $tyvars(\sigma)$ and $tyvars(TE)$ are the set of type variables that occur *free* in τ, σ and TE respectively. A type τ' is a *generic instance* of a type scheme $\sigma = \forall \alpha_1, ..., \alpha_n.\tau$, written as $\tau' \prec \sigma$, if there exists a substitution S with its domain being a subset of $\{\alpha_1, ..., \alpha_n\}$ and $\tau' = S(\tau)$. A type scheme σ_1 is more general than σ_2, denoted as $\sigma_2 \prec \sigma_1$, if all generic instances of σ_2 are also generic instances of σ_1. The generalization of a type τ in a type environment TE is denoted by $gen(TE, \tau)$, it is the type scheme $\forall \alpha_1, ..., \alpha_n.\tau$ where $\{\alpha_1, ..., \alpha_n\} = tyvars(\tau) \setminus tyvars(TE)$. The core ML type system, in the form of type deduction rules as $TE \vdash e : \tau$, is omitted here because it is the same as in Tofte [32].

2.1 The assumption inference algorithm W*

We define a *type assumption* to be a pair (x,τ) where x is a program variable and τ is a type. An *assumption environment*, ranged over by A, is a set of type assumptions; it is usually represented by a finite mapping from program variables to lists of types. In the following, we use $A \setminus \{x\}$ to denote the set of type assumptions in A except those for variable x, and $A(x)$ to denote the set of types associated with variable x in A. We also use *Unify* to denote Robinson's original unification algorithm on classical term algebras [26]. *Unify* takes a set of pairs of types and returns a substitution (the most general unifier).

Figure 1 gives the assumption inference algorithm W^* and

the matching algorithm *Match*. W^* takes an ML expression, and returns a type and an assumption environment. *Match* takes an ML type environment and an assumption environment, and returns a substitution. The other two procedures in figure 1 are *MonoUnify* and *PolyUnify*. *MonoUnify* takes a type and a set of types, and returns a substitution. *PolyUnify* takes two arguments: a triple of a TyVar set and a type and an assumption environment, and a set of types; it returns a substitution and an assumption environment.

Given an ML expression e, $W^*(e)$ delays the type-checking of all free variables in e by recording their monomorphic type instances in an assumption environment A. In the case of lambda abstraction $\lambda x.e_1$, the type α of x is treated as monomorphic; the procedure *MonoUnify* checks whether the set of assumptions collected for x from e_1 satisfies this constraint. One the other hand, in the `let` expression, the type of x is treated as polymorphic; for each use of x in e_2, the type and the assumption environment from e_1 is renamed with new type variables; the procedure *PolyUnify* then checks the typing of x in e_2 and merges the assumption environments collected from e_1 and e_2. When the real type environment TE for the free variables is known (at link time), the matching algorithm $Match(TE, A)$ precisely recovers everything, including the result type of elaborating e in TE.

For example, given an expression

$$e = \texttt{let } g = \lambda x.fx \texttt{ in } g\ g,$$

the free variable of e is f; $W^*(e)$ will return an assumption environment $A = \{f \mapsto (\alpha_6 \to \alpha_7) \to \alpha_8,\ f \mapsto (\alpha_6 \to \alpha_7),\ f \mapsto (\alpha_1 \to \alpha_3)\}$ and a type α_8 for e. If the real type environment TE is $\{f \mapsto \forall\alpha.\alpha \to \alpha\}$, $Match(TE, A)$ will result in the substitution $S^* = \{\alpha_6 \mapsto \beta_1,\ \alpha_7 \mapsto \beta_1, \alpha_1 \mapsto \beta_1,\ \alpha_3 \mapsto \beta_1,\ \alpha_8 \mapsto (\beta_1 \to \beta_1)\}$. Thus the expression e will have the type $S^*(\alpha_8) = (\beta_1 \to \beta_1)$. This is exactly what we will get if we apply Tofte's algorithm W to TE and e.

In fact we can show that the algorithm W^* is equivalent to Milner's W [32] in the following sense:

Theorem 2.1 *Given a type environment TE and an expression e, then $(S, \tau) = W(TE, e)$ succeeds if and only if both $(\tau^*, A) = W^*(e)$ and $S^* = Match(TE, A)$ succeed; Moreover, there exists two substitutions R_1 and R_2, such that the following are true: (1) $R_1 \circ R_2 = R_2 \circ R_1 = ID$; (2) $R_1(S(TE), \tau) = (S^*(TE), S^*\tau^*)$; (3) $(S(TE), \tau) = R_2(S^*(TE), S^*\tau^*)$.*

Proof By structural induction on the expression e. For details, see the technical report [30]. **QED.**

Notice that theorem 2.1 is not trying to show the soundness and completeness of W^* directly. It is just proving that the result of W^* and *Match* is equivalent to the result of W. Proving this kind of equivalence is relatively easier. From the soundness and completeness of the algorithm W(which is proved in Damas's Ph.D thesis [6]), and the above theorem 2.1, we can easily get the following soundness and completeness results for our algorithm W^*.

Corollary 2.2 (Soundness of W^*) *Given a type environment TE and an ML expression e, if both $(\tau^*, A) = W^*(e)$ and $S^* = Match(TE, A)$ succeed, then $S^*(TE') \vdash e : S^*\tau^*$.*

Corollary 2.3 (Completeness of W^*) *Given a type environment TE and an ML expression e, suppose that $TE_1 = S_1(TE)$ and $TE_1 \vdash e : \tau_1$, then both $(\tau^*, A) = W^*(e)$ and $S^* = Match(TE, A)$ will succeed; Moreover there exists a substitution S' such that $TE_1 = S'(S^*(TE))$ and $\tau_1 \prec S'(\text{gen}(S^*(TE), S^*\tau^*))$.*

The algorithm W^* itself is interesting. Recursive calls to W^* in the algorithm will not interfere with each other so they can be called in any order. If concurrency is used, W^* can be efficiently implemented. The case for the `let x = e1 in e2` expression implies that we can link two pieces of programs, i.e., e_1 and e_2, even though both of them contain free variables; this is done by the algorithm *PolyUnify* in figure 1.

The assumption environment A returned from W^* may be big. A possible optimization is to insert a simplifying procedure at each recursive call to W^* in the algorithm. This simplifying procedure will identify all "isolated" type assumptions in A. Given $(\tau, A) = W^*(e)$, we define an equivalence relation $\sim$ on type variables: "$\alpha \sim \beta$ if there exists a type τ' such that either $\tau' = \tau$ or $(x, \tau') \in A$ for some x is true, and both α and β are type variables of τ'." Let TV be the transitive closure of $tyvars(\tau)$ under $\sim$, then all pairs (x, t) in A where $tyvars(t) \cap TV = \emptyset$ are denoted as "isolated" assumptions. All type variables occurred in "isolated" assumptions need not to be renamed in *PolyUnify* and most redundant "isolated" assumptions can be eliminated.

2.2 The assumption inference algorithm D

One disadvantage of W^* is that its sequential implementation may be not very efficient in practice. In most compilers, there is a pervasive basis (or initial library) which tends to be referenced very frequently by user programs, thus the resulting assumption environment from W^* may be quite big (even if it uses certain optimizations mentioned above). It turns out that this problem can be elegantly solved by extending ML types with type predicates and assumptions. This extension, which is called "constrained type" in Kaes [13] and "qualified type" in Jones [12], is normally used to reason about the ML type system in the presence of overloading and subtyping. The algorithm D, which is presented in the appendix at the end of this paper, is the type reconstruction algorithm for Kaes's constrained type system. By using a special set of type predicates, the algorithm D can efficiently solve the assumption inference problem even when there is a pervasive basis. For more details, please see the appendix.

3 Assumption Inference in the SML Module Language

In this section, we present an assumption inference algorithm for the SML module language. To simplify the presentation, we only consider the skeletal language *ModL* (as in Tofte [32]) in figure 2. Notice that signature expressions and declarations are intentionally left out because their elaborations can be delayed to link time, thus are irrelevant to our assumption inference. Functor declarations are not considered in our language either because only their body, which is a structure expression, is elaborated at compile time. However, functor applications are considered in our language because they are structure expressions which must be elaborated at compile time.

dec	::=		m	$\in$ StrName
	|	*strdec*	N	$\in$ NameSet = Fin(StrName)
	|	*strdec dec1*	GE	$\in$ SigEnv = SigId $\xrightarrow{\text{fin}}$ Sig
			FE	$\in$ FunEnv = FunId $\xrightarrow{\text{fin}}$ FunSig
strexp	::=	*strid*	SE	$\in$ StrEnv = StrId $\xrightarrow{\text{fin}}$ Str
	|	**struct** *dec* **end**	S or (m, E)	$\in$ Str = StrName $\times$ Env
	|	*strexp.strid*	E	$\in$ Env = StrEnv
	|	*fctid(strexp)*	Σ or $(N)S$	$\in$ Sig = NameSet $\times$ Str
			$N(S, N'(S'))$	$\in$ FunSig = NameSet $\times$ (Str $\times$ Sig)
strdec	::=	**structure** *strid* = *strexp*	B	$\in$ Basis = Nameset $\times$ SigEnv $\times$ FunEnv $\times$ Env

Figure 2: *left:* Grammar; *right:* Semantic objects

The static semantics of *ModL* is discussed in detail in the definition [20] and Tofte [32]. Its deduction rule is in the form of "$B \vdash$ *phrase* $\Rightarrow A$" meaning that *phrase* is elaborated into a semantic object A in the basis B. The semantic objects are also defined in figure 2. Here we give a quick review on notations and concepts used in the static semantics: A *structure* S is a pair (m, E), where m is the name of the structure and E is an environment, which gives the static information about the components of the structure. To make the presentation clear, from now on, we shall use $\mathtt{str}(m,E)$ to denote a structure (m,E). A *signature* is an object of the form $(N)S$, where S is a structure and N is a finite set of names. A *functor signature* Φ is an object of the form $N(S, N'(S'))$ where $N(S)$ is the principal signature for the parameter signature expression of the functor and S' is the body structure of the functor, the names bound in S' are the names in S' which have to be generated afresh upon each functor application. A *structure environment SE* is a finite map from structure identifiers to structures, similarly for *signature environment GE* and *functor environment FE*. StrName is an infinite set of names: names that are specified in a signature expression and are not shared with already declared structures are called *flexible names*, denoted as FlexStrName; names of declared structures are called *rigid names*, denoted as RigStrName.

Definition 3.1 *A* realization *is a finite mapping from the set* FlexStrName *to the set* StrName*; a* renaming realization $\varphi=\{m_i \mapsto m'_i$ *where* $i = 1, ..., k\}$ *is a realization where* $m'_i s$ *are distinct rigid names.*

Definition 3.2 *A structure* $S_1 = \mathtt{str}(m_1, SE_1)$ enriches *a structure* $S_2 = \mathtt{str}(m_2, SE_2)$ *if* $m_1 = m_2$ *and the structure environment* SE_1 *enriches the* SE_2*. A structure environment* SE_1 *enriches a structure environment* SE_2 *if* $\mathrm{Dom}(SE_2) \subseteq \mathrm{Dom}(SE_1)$ *and for each* $s \in \mathrm{Dom}(SE_2)$, $SE_1(s)$ *enriches* $SE_2(s)$.

Definition 3.3 *A structure* S' *matches a signature* $\Sigma = N(S)$ *if there exists a realization* φ *such that* S' *enriches* $\varphi(S)$.

Because the SML module language is explicitly typed, the elaboration of a module expression simply involves type-checking. The static semantics in the Definition [20] can be viewed as a type checking algorithm. Given a structure expression with free identifiers, we want to infer the minimum constraints on these free identifiers with which the expression will just type-check. Again the minimum constraints are not expressible if we only use semantic objects in figure 2. We introduce a new kind of *structure variable* which is similar to row variables used in typing record calculi. Let StrVar be an infinite set of structure variables; structures and structure environments are now extended as Str′ = (StrName $\times$ StrEnv′) $\cup$ StrVar and StrEnv′ = StrId $\rightarrow$ Str′. Each structure variable t must have a kind. Kinds are defined as (StrName $\times$ KindEnv) where KindEnv is just a finite mapping StrId $\rightarrow$ (Str $\cup$ StrVar). To distinguish it from structures, a kind (m, KE) is represented as $\mathtt{STR}(m, KE)$. A kind assignment is a finite mapping from structure variables to kinds. A structure $S = \mathtt{str}(m, SE)$ has the kind k under the kind assignment K, written as $K \vdash S :: k$, if it is derivable from the following set of kinding rules:

(1) $K \vdash t :: \mathtt{STR}(m,KE)$, if $K(t) = \mathtt{STR}(m, KE)$
(2) $K \vdash \mathtt{str}(m, SE) :: \mathtt{STR}(m, KE)$
if both $\mathrm{Dom}(SE) \supseteq \mathrm{Dom}(KE)$
and $\forall \mathtt{s} \in \mathrm{Dom}(KE), SE(\mathtt{s}) = KE(\mathtt{s})$.

A substitution now consists of two parts: one from StrVar to Str′, another from FlexStrName to StrName (i.e., realization). A kinded substitution is a pair consisting of a kind assignment and a substitution. A kinded substitution (K_1,R) respects a kind assignment K_2 if, for all t in $\mathrm{dom}(K_2)$, $K_1 \vdash R(t) :: R(K_2(t))$ is a derivable kinding. A kinded substitution (K_1,R_1) is more general than (K_2,R_2) if $R_2 = R_3 \circ R_1$ for some R_3 such that (K_2,R_3) respects K_1. A kinded substitution (K_1,R) is an unifier of a kinded set of equations (K_2,P) if it respects K_2 and $R(t_1) = R(t_2)$ for all (t_1, t_2) in P.

Figure 3 gives our inference algorithms W_{strexp} on structure expressions, W_{dec} on declarations, W_{fctid} on functor identifiers, W_{strdec} on structure declarations and the matching algorithm *ModlMatch*. The argument V and M records those already-used structure variables and flexible names. All functor applications are done by the matching algorithm at link time. The "thinning effect" in functor applications (on the argument signature) is achieved by the set of constraints generated by the *GenRec* algorithm. The inferred assumption environment A automatically records the "min-

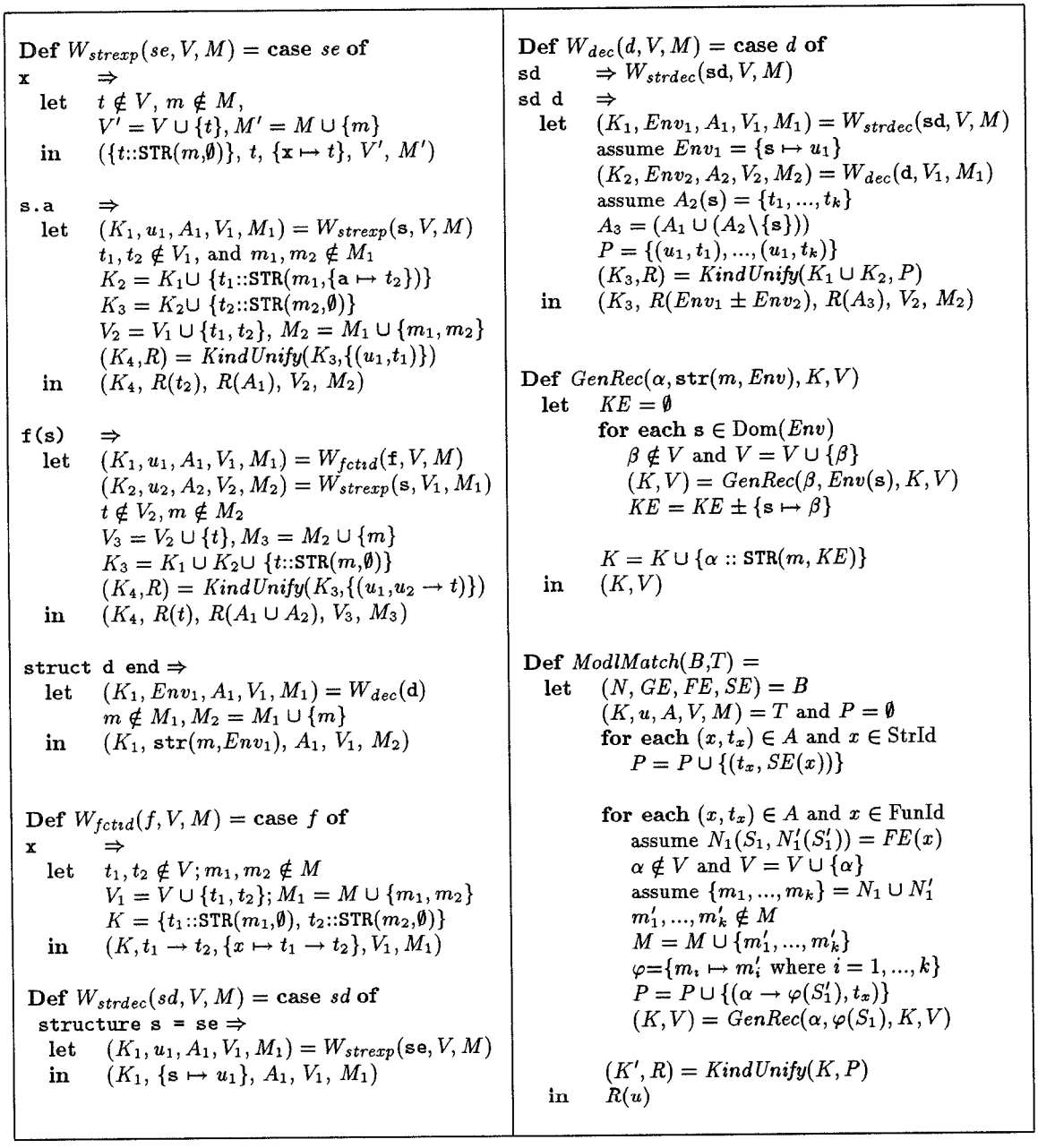

Figure 3: Assumption Inference in *ModL*

Def $KindUnify(K, P) =$ **case** (K, P) **of**

$(K_1, \emptyset) \Rightarrow (K_1, ID)$
$(K_1, P_1 \cup \{(t, t)\}) \Rightarrow KindUnify(K_1, P_1)$
$(K_1, P_1 \cup \{(t_1 \rightarrow t_2, t_1' \rightarrow t_2')\}) \Rightarrow KindUnify(K_1, P_1 \cup \{(t_1, t_1'), (t_2, t_2')\})$

$(K_1 \cup \{t_1 :: \mathtt{STR}(m_1, Env_1)\}, P_1 \cup \{(t_1, \mathtt{str}(m_2, Env_2))\}) \Rightarrow$
  let check $\mathrm{Dom}(Env_1) \subseteq \mathrm{Dom}(Env_2)$, otherwise fail
    $(R_m, m) = NameUnify(m_1, m_2)$ and $Env_1' = R_m(Env_1)$ and $Env_2' = R_m(Env_2)$
    $R = (\{t_1 \mapsto \mathtt{str}(m, Env_2')\})(R_m)$ and $K_2 = R(K_1)$
    $P_2 = R(P_1) \cup \{(Env_1'(s), Env_2'(s)) \mid \forall s \in \mathrm{Dom}(Env_1)\}$
    $(K_3, R') = KindUnify(K_2, P_2)$
  in $(K_3, R' \circ R)$

$(K_1 \cup \{t_1 :: \mathtt{STR}(m_1, Env_1), t_2 :: \mathtt{STR}(m_2, Env_2)\}, P_1 \cup \{(t_1, t_2)\}) \Rightarrow$
  let $(R_m, m) = NameUnify(m_1, m_2)$ and $R = (\{t_1 \mapsto t_2\})(R_m)$
    $Env_1' = R(Env_1)$ and $Env_2' = R(Env_2)$ and $Env' = Env_2' \cup (Env_1' \setminus \mathrm{Dom}(Env_2'))$
    $K_2 = R(K_1) \cup \{t_2 :: \mathtt{STR}(m, Env')\}$
    $P_2 = R(P_1) \cup \{(Env_1'(s), Env_2'(s)) \mid \forall s \in \mathrm{Dom}(Env_1') \cap \mathrm{Dom}(Env_2')\}$
    $(K_3, R') = KindUnify(K_2, P_2)$
  in $(K_3, R' \circ R)$

$(K_1, P_1 \cup \{(\mathtt{str}(m_1, Env_1), \mathtt{str}(m_2, Env_2))\}) \Rightarrow$
  let check $\mathrm{Dom}(Env_1) = \mathrm{Dom}(Env_2)$, otherwise fail
    $(R_m, m) = NameUnify(m_1, m_2)$ and $Env_1' = R_m(Env_1)$ and $Env_2' = R_m(Env_2)$
    $K_2 = R_m(K_1)$ and $P_2 = R_m(P_1) \cup \{(Env_1'(s), Env_2'(s)) \mid \forall s \in Dom(Env_1')\}$
    $(K_3, R') = KindUnify(K_2, P_2)$
  in $(K_3, R' \circ R_m)$

Def $NameUnify(m_1, m_2) =$
    if $m_1 = m_2$ **then** (ID, m_1)
    else if $m_1, m_2 \in$ RigStrName **then** *fail*
    else if $m_1 \in$ RigStrName **then** $(\{m_2 \mapsto m_1\}, m_1)$
    else $(\{m_1 \mapsto m_2\}, m_2)$

Figure 4: Kinded unification algorithm

imum" sharing constraints required to make the structure expression elaborate.

Figure 4 gives the unification algorithms *KindUnify* and *NameUnify*. The kinded unification algorithm *KindUnify* presented there extends the one in Ohori [24] with considerations on ML structure names. The following theorem can be proved in the same way as Ohori [24].

Theorem 3.1 *Given any kinded set of equations, the algorithm KindUnify computes a most general unifier if one exists and reports failure otherwise.*

The following lemma shows how the "thinning" effect is achieved in our algorithm.

Lemma 3.2 *Given a signature* $\Sigma = N(S)$ *and a structure* S'*, suppose that* S' *does not contain any flexible names; let* α *be a structure variable and* V_1 *be any set of structure variables; suppose that* $(K, V) = GenRec(\alpha, S, \emptyset, V_1)$*, then* $KindUnify(K, \{(\alpha, S')\})$ *succeeds if and only if the structure* S' *matches the signature* Σ*. Moreover, if* $R = KindUnify(K, \{(\alpha, S')\})$*, then* S' *enriches* $R(S)$*.*

The following theorem can be proved by structural induction on structure expressions.

Theorem 3.3 *Given an* ML *basis* B *and a structure expression strexp, then* $B \vdash strexp : S$ *succeeds if and only if both* $(K, u, A, V, M) = W_{strexp}(strexp, \emptyset, \text{RigStrName})$ *and* $S' = ModlMatch(B, (K, u, A, V, M))$ *succeed. Moreover, there exists a renaming realization* φ *such that* $S = \varphi(S')$*.*

The algorithm W_{strexp} possesses most properties that W^* has. It can also be modified to take a basis B as its argument (just as algorithm D) so that it can work more efficiently when we compile a structure expression in the pervasive basis.

4 Code Generation Issues

A compiling process usually contains two parts: elaboration (i.e., type inference or type-checking) and code generation (also code optimization). The assumption inference algorithms presented in the last two sections successfully solve the problem in the elaboration phase. In order to achieve the "smartest recompilation rule", our compiler should generate code that will be reusable as long as the surroundings satisfy the "minimum" import interface (i.e., match the assumption environment). This requires that our code generator should use no more type information than is specified in the "minimum" import interface.

Fortunately there are very few dependencies between the static semantics and the dynamic semantics in SML. Moreover, although Leroy's representation analysis [17] shows that the compiler can benefit a lot by using type information in the front end, the SML/NJ compiler [4] uses almost no type information in its back end but it still produces quite efficient code. In SML/NJ, the only things that the back end needs to know from the front end are the corresponding dynamic interface for each signature and the identifier status for each identifier. By delaying these dependencies to be resolved at link time, a program can be translated into machine code even before it is elaborated.

Because of space limitations, we only informally discuss the solutions to these issues here.

Functor application

In SML, a functor F with argument signature *SIG* can be applied to any structure S that *matches SIG*. A structure does not have to agree exactly with a signature in order for it to match the signature, instead it can contain more components than required. In such cases, signature matching will coerce the structure against the signature, producing a "thinned" structure that exactly agrees with the signature in terms of number of components and their types. Suppose the corresponding dynamic code for the functor F and the structure S is *fd* and *sd*, the code generated for a function application $F(S)$ will be $fd(th(sd))$ where *th* is a thinning function from S to *SIG*. In our separate compilation scheme, it is possible that we still do not know the argument signature of F or the exact specification of structure S when we have to generate code for the functor application $F(S)$. This is simply solved by adding an abstraction on *th*, and the code becomes $\lambda th.(...fd(th(sd))...)$. The correct thinning function is filled in at link time when the real specifications of *SIG* and S are known.

Pattern matching

In the SML/NJ compiler, the representation of a user defined datatype is determined by its definition. For example,

```
structure A
  = struct
      datatype color = RED | GREEN | BLUE
                     | MIX of real * real * real
    end

structure B
  = struct fun redp(A.RED) = 1.0
             | redp(A.MIX(x,_,_)) = x
             | redp _ = 0.0
    end
```

the datatype `color` in `A` may be represented with integer tags 0,1,2,3 for the data constructors `BLUE`, `GREEN`, `MIX`, and `RED`. However this imposes some problems if we want to separately compile structure `B`. What representations are we going to use for `A.RED` and `A.MIX` in the `redp` function? Again this is solved by making the representation of data constructors abstract (as in Aitken and Reppy's recent work [2]). A "constant" data constructor (such as `A.RED`) is compiled as a variable. A value carrying constructor (such as `A.MIX`) is compiled as a pair of injection and projection functions. These details are filled in at link time when the definition of the datatype is known.

Polymorphic equality function

Nothing needs to be done to support our separate compilation scheme if the equality function is implemented as it currently is in the SML/NJ compiler. In SML/NJ (as in all ML compilers to our knowledge), the polymorphic equality function is implemented as a runtime "equality interpreter" which checks equality of two objects based on their runtime tags. Another way to implement polymorphic equality, which is used in Haskell [11], is to pass an equality function for each formal parameter that is a polymorphic equality type variable. The code produced by this scheme closely depends on the derivation tree of the elaboration phase. In our separate compilation scheme, because the types of

some external identifiers are not known at compile time, the derivation tree we get at compile time is not accurate. For example,

```
fun f x = S.g (3,x)
```

from assumption inference, we know `S.g`'s type must be in the form of $int * \alpha \rightarrow \beta$ and `f`'s in the form of $\alpha \rightarrow \beta$. Because `S.g` may want to test the equality on its 2nd argument, the function `f` here has to be implemented with an equality function for type α as its extra argument. This will have some runtime overhead in the common case that `S.g` actually never does equality test on its 2nd argument.

Representational analysis

Leroy [17] presented a program transformation that allows polymorphic languages to be implemented with unboxed, multi-word data representation. The main idea is to introduce coercions between various representations based on the typing derivation tree. In our separate compilation system, accurate type information for external identifiers are not available at compile time, so the typing derivation tree is not very specific. However the representation analysis can still be carried out since all type instances of external identifiers are recorded in the assumption environment. At link time, the matching algorithm *Match* will find out the accurate type information of all external identifiers and coerce them into different type instances in the assumption environment. For example, the above function `f` will be implemented as a polymorphic function $\alpha \rightarrow \beta$. When we find that `S.g` has type $int * int \rightarrow int$, `S.g` has to be coerced to type $int * \alpha \rightarrow \beta$ and `f` has to be coerced from $\alpha \rightarrow \beta$ to $int \rightarrow int$. The code produced in this way will be less efficient, but it should be acceptable if in practice there are not too many external identifiers in a module (especially when we use algorithm D).

Open declaration

The open declaration in SML causes several nasty problems for our separate compilation scheme. We have solved these problems by delaying certain operations to link time [30]. Our solutions may increase the complexity of linking but they do not incur any runtime overhead (however, they may stop some inline-expansion optimizations).

5 Implementation

We are currently prototyping a separate compilation system based on our algorithms into the SML/NJ compiler. In our system, a large ML program is composed of a set of top-level structure declarations, signature declarations and functor declarations. No two top-level structures (or signatures, functors) can have the same identifier name so that we can uniquely determine which definition each external identifier refers to. Every top-level declaration is considered as a compilation unit. Because signatures are usually small and compiling signature declarations does not take much time, their elaborations are delayed to be done at link time. To compile a structure or functor declaration, we apply the assumption inference algorithm to its body (which is always a structure expression), generate the machine code for the body, and then write both the inferred interface (i.e., the assumption environment) and the machine code into its binary file. The final linking phase is done in certain order according to the dependency relation among different modules. This dependency relation has been already recorded in the inferred interface in each binary file. For each module, the linker simply reads the binary file, elaborates every signature expression, applies the matching algorithm (i.e., *Match* and *ModlMatch*) to recover the correct static environment and detect cross-module type errors if there are any, and then concatenates the machine code with correct thinning functions.

6 Related Work

Most dynamically-typed languages such as Lisp also allow independent compilations and can achieve the same kind of "smartest recompilation" in the sense that a module never needs to be recompiled unless its implementation changes. However, this is based on a big sacrifice: cross-module type errors will be detected only at runtime. Our method, however, will detect all cross-module type errors at link time.

Levy [18] presents a separate compilation method very similar to ours for PASCAL-like languages. Its compiler also automatically infers the import interface for each compilation unit. Cross-module type errors are reported at link time. However he does not mention whether he achieves the smartest recompilation rule, and the type systems of PASCAL-like languages are much simpler than that of SML.

Traditional separate compilation systems adopted in most statically typed languages all use manually created interface files. Each compilation unit contains an implementation plus several interface files. It has to be fully closed up to the pervasive basis so that the specifications of all external symbols will be found at compile time. The `make` system [9] is the simplest one along this line. It will trigger recompilations if the interface file a module depends on changes. Tichy [31] and Schwanke [29] eliminated most recompilations by examining finer-levels of dependency relations between interfaces and implementations. In their methods, if the interface file a module depends on changes, but the set of symbols the module imports does not change, then the module does not need to be recompiled. SRC Modula-3 [22, 14] implements exactly the same idea: a version stamp which encodes the specification of a symbol is produced for each exported symbol in an interface; modules import the version stamps of the symbols that they import; a module only needs to be recompiled if any of its imported version stamps are no longer exported. Languages with very powerful module systems such as Mesa [21], the System Modeller in Cedar [15], and FX-87 [8] also adopt similar separate compilation methods which are only applicable to closed modules. The compiler for Russell [5] does partially support separate compilations on "open-formed" expressions, however its "module system" is very restrictive and all "modules" must be loaded and compiled in an order determined by their dependencies. In summary, these previous methods cannot achieve the smartest recompilation rule, neither can they be applied to compile open-formed modules in SML.

In SML, two kinds of separate compilation methods have been proposed: Rothwell and Tofte's `import` scheme [28] and Rollins's `SourceGroup` scheme [27]; both methods apply only to closed functors. Recently, Emden Gansner [10]

is implementing a make-like separate compilation system for open-formed modules in the interactive SML/NJ compiler. In his method, all modules are loaded and compiled in a top level environment in an order determined by their dependencies. Whenever a module is compiled, a new time stamp is generated; both the binary and the time stamp are then written out to the binary file. A module has to be recompiled whenever its source changes or any of its predecessors (in the dependency graph) have been recompiled. Gansner is also planning to export the static semantics of each module into the binary file so that redundant recompilations can be detected and avoided if the static semantics of a module has not been changed.

Aditya and Nikhil [1] have been working on similar kinds of assumption inference algorithms for their incremental compiler for Id [23]. However as far as we know, their algorithm does not infer the minimum constraints, thus fails to achieve our theorem 2.1. Because their system allows mutually recursive top-level declarations, it cannot fully recover the correct type information by simply using our assumption inference and matching algorithm. In the SML module language, however, top-level declarations cannot be mutually recursive.

Damas [6] gave an inference algorithm called T which is very similar to our W^* in section 2. His type system permits that a variable can be bound to several distinct types in the type environment (just like our assumption environment). However since he mainly used the system to handle overloading, he did not try to prove our theorem 2.1. The soundness and syntactic completeness results he proved for T are only for his particular type system, not for the usual ML type system [32], so they are not in the same sense as our corollary 2.2 and 2.3. The algorithm V in Leivant [16] is just Damas's T restricted to the type system without ML-polymorphism. Its extension V_2 is for the polymorphic discipline of rank 2 and the relation between W and V_2 is not clear. On the side of the SML module language, Aponte [3] presented a type checking algorithm for *ModL* based on Remy's approach to record typing [25]. Her approach is very elegant; however, in practice it is probably very difficult to implement efficiently. It is also not clear whether her algorithm can be modified to do our assumption inference.

7 Concluding Remarks

We have presented a separate compilation method that achieves the "smartest recompilation rule" for open-formed modules in Standard ML. In our method, each module is compiled independently without knowing the specifications of its external identifiers; its import interface, instead, is inferred by looking at how each external identifier is used inside the module. Cross-module type inconsistencies are detected at link time by simply matching the real specifications against the inferred import interface (this process should be very fast because it only involves an unification of a set of types). The independent compilation of each module may disable some inter-module optimizations, but we believe that the code generated by our recompilation method will be comparable to the quite efficient code generated by the current SML/NJ compiler [4]. We plan to implement and measure our algorithm in SML/NJ in the future.

The smartest recompilation technique in this paper is presented in the framework of SML; however, it can be easily applied to other polymorphic languages based on the Damas-Milner type discipline. It should be straightforward to extend the algorithm D in section 2.2 to work on the extension of ML type system with parametric overloadings [13]. The assumption inference algorithms presented in section 2 and 3 can also be used as a basis to build incremental compilers for similar languages. On the other hand, we still do not know how to extend the algorithm for *ModL* to work on the extension of ML module system with high-order functors [33].

The smartest recompilation technique should also be applicable to languages in the Algol family. The type system in those languages are much simpler than that in ML, so it is not hard to infer the "minimum" import interface for each module. However, the code produced by smartest recompilation will be less efficient because the code generators of these languages usually rely much more on the inter-procedure type and data flow information than those of polymorphic languages. For applications where reusability and reconfiguration are more important than efficiency, the smartest recompilation property is still very desirable.

Acknowledgements

We would like to thank William Aitken, Carl Gunter, and QingMing Ma for many valuable comments on an early version of this paper. We are also grateful to David MacQueen and Pierre Cregut for interesting discussions on related subjects. This research is supported by the National Science Foundation Grant CCR-9002786 and CCR-9200790, and by the first author's summer research internship at AT&T Bell Laboratories.

References

[1] Shail Aditya and Rishiyur S. Nikhil. Incremental polymorphism. In *The Fifth International Conference on Functional Programming Languages and Computer Architecture*, pages 378–405, New York, August 1991. Springer-Verlag.

[2] William E. Aitken and John H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, June 1992.

[3] Maria Virginia Aponte. *Typage d'un systeme de modules parametriques avec partage: une application de l'unification dans les theories equationnelles.* PhD thesis, Universite de Paris, February 1992.

[4] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[5] Hans Boehm and Alan J. Demers. Implementing Russell. In *Symposium on Compiler Construction*, pages 186–195. ACM Sigplan, June 1986.

[6] Luis Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, Department of Computer Science, Edinburgh, UK, 1985.

[7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1982. ACM Press.

[8] David K. Gifford et al. FX-87 reference manual. Technical Report MIT/LCS/TR-407, M.I.T. Laboratory for Computer Science, September 1987.

[9] Stuart I. Feldman. Make – a program for maintaining computer programs. *Software – Practice and Experience*, 9(4):255–265, Apirl 1979.

[10] Emden R. Gansner. AT&T Bell Labs, personal communication, 1992.

[11] Paul Hudak, Simon Peyton Jones, and Philip Wadler *et al.* Report on the programming language Haskell a non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 21(5), May 1992.

[12] Mark P. Jones. A theory of qualified types. In *The 4th European Symposium on Programming*, pages 287–306, Berlin, February 1992. Spinger-Verlag.

[13] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *1992 ACM Conference on Lisp and Fucntional Programming*, New York, June 1992. ACM Press.

[14] Bill Kalsow and Eric Muller. SRC Modula-3 version 1.6 manual, February 1991.

[15] Butler W. Lampson and Eric S. Schmidt. Pratical use of a polymorphic applicative language. In *Tenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1983. ACM Press.

[16] Daniel Leivant. Polymorphic type inference. In *Tenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1983. ACM Press.

[17] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

[18] Michael R. Levy. Type checking, separate compilation and reusability. *SIGPLAN Notices (Proc. Sigplan '84 Symp. on Compiler Construction)*, 19(6):285–289, June 1984.

[19] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Cambridge, Massachusetts, 1991.

[20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[21] J. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA, 1979.

[22] Greg Nelson, editor. *Systems programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[23] Rishiyur S. Nikhil. Id version 90.0 reference manual. Technical Report TR-CSG-Memo 284-1, MIT Laboratory for Computer Science, 1990.

[24] Atsushi Ohori. A compilation method for ML-style polymorphic record calculi. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

[25] Didier Remy. Typechecking records and variants in a natural extension of ML. In *Sixteenth Annual ACM Symp. on Principles of Prog. Languages*, pages 77–87, New York, Jan 1989. ACM Press.

[26] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[27] Eugene J. Rollins. SourceGroup: A selective recompilation system for SML. In *Third International Workshop on Standard ML*, Pittsburgh, September 1991. Carnegie Mellon University.

[28] Nick Rothwell and Mads Tofte. **Import** command source code. with Standard ML of New Jersey releases 0.65.

[29] Robert W. Schwanke and Gail E. Kaiser. Smarter recompilation. *ACM Transactions on Programming Languages and Systems*, 10(4):627–632, October 1988.

[30] Zhong Shao and Andrew W. Appel. Smartest recompilation. Technical Report CS-TR-395-92, Princeton Univ. Dept. of Computer Science, Princeton, NJ, October 1992.

[31] Walter Tichy. Smart recompilation. *ACM Transactions on Programming Languages and Systems*, 8(3):273–291, July 1986.

[32] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, Edinburgh, UK, November 1987.

[33] Mads Tofte. Principal signatures for high-order ML functors. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

8 Appendix: the assumption inference algorithm D

8.1 An extension of ML with constrained types

The extension of ML type system with constrained types (denoted as ML^+) is discussed in detail by Kaes [13] and Jones [12] to solve the type inference problem in languages that support overloading and subtyping. It turns out that it can also be used to solve our assumption inference problem. The language syntax they use is essentially same as the mini-ML language *Exp*. In the following, we give a quick review of Kaes's framework of extending ML with constrained types. To ease the notation, $\overline{x_n}$ is used to denote a sequence $x_1, ..., x_n$.

Definition 8.1 *Let P be a finite n-indexed family of predicate symbols. The set of* predicate constraints *over Type is defined as $\{p(\overline{\tau_n}) \mid p \in P_n, \tau_i \in Type$ where $i = 1, ..., n\}$. An interpretation of P is a family of total computable functions $(\hat{p})_{p \in P}$, such that for $p \in P_n, \hat{p} : (\text{Type})^n \to \mathbf{2}$.*

Definition 8.2 *A set C of constraints is* satisfiable *if there exists a substitution S such that if $p(\overline{\tau_n}) \in C$ then $\hat{p}(S(\overline{\tau_n}))$ is true. Satisfiablity will be denoted as $S \models C$.*

Definition 8.3 *A substitution S is a* solution *of C, if $S' \circ S \models C$ for all substitutions S'. A solution S is called the* most general solution *of a constraint set C, if for any solution R of C, there exists a substitution S', such that $R = S' \circ S$. In most cases, the most general solution does not exist.*

The *entailment* relation on constraint sets, written $C_1 \Vdash C_2$, may be defined once a particular predicate system is given. In the following, we only consider those predicate systems which satisfy the following properties: if $C_1 \Vdash C_2$ then $\forall S : S \models C_1 \Rightarrow S \models C_2$.

Definition 8.4 *A* constrained type *is a pair $\tau|C$, consisting of a type τ and a set of constraints C. A* constrained type scheme *is of the form $\forall\overline{\alpha_n}.\tau|C$. A* constrained type environment *is now just a finite map from variables to constrained type schemes.*

Definition 8.5 *A constrained type $\tau' \mid C'$ is a* generic instance *of a constrained type scheme $\sigma = \forall\overline{\alpha_n}.\tau|C$, written as $\tau'|C' \prec \sigma$, if there exists a substitution S with domain being a subset of $\overline{\alpha_n}$ such that $\tau' = S(\tau)$ and $C' \Vdash S(C)$. We also denote $\sigma' \prec \sigma$ if all generic instances of σ' are generic instances of σ; and $\sigma' \equiv \sigma$ if $\sigma' \prec \sigma$ and $\sigma \prec \sigma'$.*

Def $D(TE, e)$ = **case** e **of**

x $\Rightarrow$ **let** $TE(x) = \forall\overline{\alpha_n}.\tau|C$ and $\overline{\beta_n}$ be new type variables and $S = \{\alpha_i \mapsto \beta_i \text{ for } i = 1, ..., n\}$
in $(ID\ ,\ S(\tau)|S(C))$

$\lambda x.e_1$ $\Rightarrow$ **let** α be a new type variable and $(S_1, \tau_1|C_1) = D(TE \pm \{x \mapsto \alpha|\emptyset\}, e_1)$
in $(S_1\ ,\ (S_1(\alpha) \rightarrow \tau_1)|C_1)$

e_1e_2 $\Rightarrow$ **let** $(S_1, \tau_1|C_1) = D(TE, e_1)$ and $(S_2, \tau_2|C_2) = D(S_1(TE), e_2)$
α be a new type variable and $S_3 = Unify(S_2\tau_1, \tau_2 \rightarrow \alpha)$
in $(S_3 \circ S_2 \circ S_1\ ,\ (S_3\alpha)|(S_3(S_2(C_1) \cup C_2)))$

`let` x `=` e_1 `in` e_2 $\Rightarrow$
let $(S_1, \tau_1|C_1) = D(TE, e_1)$ and $(S_2, \tau_2|C_2) = D(S_1(TE) \pm \{x \mapsto gen(S_1(TE), \tau_1|C_1)\}, e_2)$
in $(S_2 \circ S_1\ ,\ \tau_2|(S_2(C_1) \cup C_2))$

Figure 5: The Type Inference Algorithm D

Typability of an expression e in ML$^+$ is expressed as a judgement $C, TE \vdash e : \tau$, which can be read as "e has type τ under (constrained) type environment TE, provided C is satisfiable." The generalization of a constrained type $\tau \mid C$ in the context of a type environment TE is denoted by $gen(TE, \tau \mid C)$, it is the constrained type scheme $\forall\overline{\alpha_n}.\tau|C'$ where $\{\alpha_1, ..., \alpha_n\} = tyvars(\tau|C) \setminus tyvars(TE)$ and $C' = \{p(\overline{\tau}) \in C \text{ where } tyvars(p(\overline{\tau})) \cap \overline{\alpha_n} \neq \emptyset\}$.

Definition 8.6 *A typing $C, TE \vdash e : \tau$ is* more general *than $C', TE' \vdash e : \tau'$, if there exists a substitution S, such that (1) $x \in dom(TE) \Rightarrow TE'(x) \prec S(TE(x))$ (2) $gen(TE', \tau' \mid C') \prec S(gen(TE, \tau|C))$.*

Kaes [13] presented the type deduction rules (also listed in the appendix at the end of this paper for reference) and the type inference algorithm D (as in figure 5) for the above extension. It can be proved that his type inference algorithm D is sound and (syntactically) complete in the following sense:

Theorem 8.1 *Let an instance of a constraint based inference system be given, e be an expression, TE and TE' be type environments. Suppose for certain substitution S_1, for each $x \in dom(TE)$, $TE'(x) \prec S_1(TE(x))$; and $C', TE' \vdash e : \tau'$ is a valid typing, then $(S, \tau \mid C) = D(TE, e)$ succeeds. Moreover, $C, S(TE) \vdash e : \tau$ is valid and more general than $C', TE' \vdash e : \tau'$.*

8.2 Application to assumption inference

We can use the above extension to solve our assumption inference problem. The set of predicates we use, denoted by P_m, is $\{p_x(\tau)$ where x is any program variable$\}$. The interpretation of p_x is "$\hat{p_x}(\tau)$ = true if and only if $\tau \prec \sigma$, assuming that the type of x is a closed ML type scheme σ." The entailment relation on constraint sets is defined as: $C_1 \Vdash C_2$ if and only if C_1 is satisfiable and $\forall S : S \models C_1 \Rightarrow S \models C_2$. This relation is decidable for our particular predicate system P_m because of the following lemma:

Lemma 8.2 *For any constraint set C formed in the predicate system P_m, either there is no solution or there exists a most general solution S.*

Proof If we consider each $p_x(\tau)$ as an assumption (x, τ), also assume that the type of x is known, the *Match* algorithm in figure 1 can be used to find the most general solution of C. The lemma then follows from Robinson's unification theorem. **QED.**

Given a closed ML type scheme $\sigma = \forall\overline{\alpha_n}.\tau$, it can be written as ML$^+$ constrained type schemes $\sigma_1 = \forall\overline{\alpha_n}.\tau|\emptyset$ or $\sigma_2 = \forall\alpha.\alpha|\{p_x(\alpha)\}$, but obviously $\sigma_1 \prec \sigma_2$ in ML$^+$.

The great thing about the algorithm D is that when it is running, it does not need any knowledge about the interpretation of the predicate system. This leads to the following theorem:

Theorem 8.3 *Given a* ML *type environment $TE = TE_1 \pm TE_2$, where $\mathrm{Dom}(TE_1) \cap \mathrm{Dom}(TE_2) = \emptyset$ and $tyvars(TE_2) = \emptyset$, we construct a* ML$^+$ *constrained type environment $TE' = TE_1' \pm TE_2'$ where $TE_1' = \{x \mapsto \forall\overline{\beta_n}.(\tau \mid \emptyset)$ where $x \in \mathrm{Dom}(TE_1)$ and $TE_1(x) = \forall\overline{\beta_n}.\tau\}$ and $TE_2' = \{x \mapsto \forall\alpha.(\alpha \mid \{p_x(\alpha)\})$ where $x \in \mathrm{Dom}(TE_2)\}$ and the interpretation of p_x is "$\hat{p_x}(\tau)$ = true if and only if $\tau \prec TE_2(x)$". Then $(S, \tau) = W(TE, e)$ succeeds if and only if $(S', \tau' \mid C') = D(TE', e)$ succeeds and there exists a most general solution S^* for C'. Moreover, there exists two substitutions R_1 and R_2, such that the following are true: (1) $R_1 \circ R_2 = R_2 \circ R_1 = ID$; (2) $R_1(S(TE), \tau) = (S^*(S'(TE)), S^*\tau')$; (3) $(S(TE), \tau) = R_2(S^*(S'(TE)), S^*\tau')$.*

Proof Follows from lemma 8.2 and theorem 8.1. For details, see the technical report [30]. **QED.**

In practice algorithm D will be more useful than W^* because it resembles the algorithm W and it also works more efficiently when the types of some free variables are known. Moreover, D can be easily extended to work on various extensions of the ML type system with overloading and subtyping such as those in Kaes [13].