Soft Interfaces: Typing Scheme at the Module Level

Martin Gasbichler Holger Gast
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen
Sand 13
D-72076 Tübingen, Germany
{gasbichl, gast}@informatik.uni-tuebingen.de

Abstract

Soft Interfaces is a module system for Scheme with typed interfaces. This makes it possible to declare the types of exported procedures in a language with union types, parametric polymorphism and recursion. Soft Interfaces statically checks calls to these procedures to ensure that the arguments meet the declared types. Implementations of exported procedures are checked to have a subtype of their declared type. Unlike other systems for reconstructing types in a dynamically typed language Soft Interfaces supports abstract types including the inference of their implementation types. We have implemented Soft Interfaces within Scheme 48.

1 Introduction

Consider a procedure that merges two sorted lists of arbitrary elements:

```
(define (merge-sorted l1 l2 elt<)
  (if (null? l1) l2
      (if (null? l2) l1
          (if (< (car l1) (car l2))
              (cons (car l1)
                    (merge-sorted (cdr l1) l2 elt<))
              (cons (car l2)
                    (merge-sorted l1 (cdr l2) elt<))))))
```

This program contains a bug: The procedure `elt<`, intended to compare the elements of the lists, is never applied. Instead, `<` is used for comparison. As long as `merge-sorted` is used in a larger program or in a test suite, but only applied to integer lists, everything will work fine despite the bug. This could lead to the misconception that the procedure is correct.

Type systems provide means to reason about all the possible values a procedure may be applied to and may produce. Several tools exist to tame dynamic typing by applying static type checkers to Scheme programs [21, 8, 1]. Among these, Soft Scheme's[21] type system is a compromise between to readability and exactness of the inferred types. Since type systems for dynamically typed languages are inherently too weak to determine all types exactly, either meaningful programs have to be rejected or type checks for applications of primitive procedures that *may* fail have to be postponed to run-time. Soft Scheme does the latter, since its main aim is the reduction of run-time checks in arbitrary Scheme programs.

Can Soft Scheme help our implementor of `merge-sorted`? The inferred type lists only integer lists as acceptable parameters for `merge-sorted`. The programmer merely has to read the output of the type inference system. This is simple here, but stops to be trivial in a large program, especially as the types inferred by Soft Scheme tend to be complicated (In his thesis [20], Wright presents the type for an expression parser, ranging over more than 30 lines). As Soft Scheme processes only entire programs, the sheer number of inferred types pushes the system to the limits of usability. Soft Scheme does emit warnings at compile time if the application of a primitive is sure to fail. When the program contains applications of `merge-sorted` to both integer lists and lists of strings the type system adds a run-time check.

There is a problem with type inference: it embodies essentially a one-way communication from the inference engine to the user – but there is no way for the programmer to tell the system anything about her intentions. Other languages like ML or Haskell support type annotations against which the inferred types are checked. With parametric polymorphism it is even possible to specify that a procedure should be applicable to any value or that there exist certain relationships between the arguments.

Imagine the possibility of declaring a type annotation for `merge-sorted`:

```
(merge-sorted ((list 'X1) (list 'X1)
           ('X1 'X1 -> bool) -> (list 'X1)))
```

This states that `merge-sorted` is a procedure with three arguments: two lists over a type variable `'X1` and a procedure taking two `'X1`s and returning a boolean value. The return value of `Merge-sorted` is a list of `'X1`s.

Imagine further a system with the possibility to check whether the implementation of `merge-sorted` obeys this declaration. From this system, we would expect an error message like this:

```
Error: Type does not match declaration:
  Inferrred: ((list num) (list num) 'X1 -> (list num))
  Declared:  ((list 'X1) (list 'X1)
              ('X1 'X1 -> bool) -> (list 'X1))
```

To summarize, we can formulate the following demands for better debugging support in Scheme:

- The user must be able to declare type annotations the implementation must obey.
- The type language must embody a good compromise between expressiveness and simplicity.
- It should be possible to type check parts of a program independently.

We use modules as units of type checking and interfaces as an appropriate place for type annotations. An advantage of this approach is that the Scheme programs themselves remain untouched, which provides backwards compatibility as well as a clear separation of implementation and specification.

However, typed interfaces carry the risk that too much information is revealed in the interface, tempting the user of the module to exploit implementation details.

Consider this implementation of environments as association lists:

```
(define (getenv a-list key)
  (let ((r (assq key a-list)))
    (if r (cdr r) #f)))
(define (extend-env alist key val)
  (cons (cons key val) alist))
(define (empty-env) '()))
```

If the type of these procedures are revealed in an interface the user has the possibility to count the number of entries by applying `length` to an environment. If some day the implementation changes to use procedures to represent environments, the code using `length` will not work any more.

Consequently, there must be a way to hide information about types in interfaces. Barbara Liskov et al. [11] have introduced the concept of abstract types to achieve this. Outside the module abstract types are regarded to be incompatible with any other type apart from itself.

Using a parameterized abstract type `env` the declarations for environments would look like this:[1]

```
(typedef (env 'X1 'X2))
(getenv ((env 'X1 (+ false 'X2)) 'X1 -> (+ false 'X2)))
(extend-env ((env 'X1 'X2) 'X1 'X2 -> (env 'X1 'X2)))
(empty-env (-> (env 'X1 'X2)))
```

We now expect a type checking error like

```
Error: Type error for application of procedure length
       Inferred type of argument 1: env
       Accepted type of argument 1: list
```

for the expression `(length (empty-env))`.

Usually, abstract types emerge from concrete types declared within the module by forgetting about the implementation; only the name remains. We do not have type declarations within the source code. Therefore a type checker for typed interfaces must be able to operate without explicit declarations of the implementation types. In the presence of union types, such reconstructions of types must be handled with care to retain useful assistance during debugging. To see why, consider two procedures, both specified to return the same abstract type. If their implementations return different types, say numbers and strings, these are typeable by taking the abstract type to be the union of numbers and strings. Only the implementor may decide, whether this is intended or an error. Therefore, she needs to see the inferred type.

We are now in the position to state our forth and final demand for better debugging support:

- The system should support abstract types and infer their implementation types.

[1] In Soft Scheme's notation + denotes union types.

1.1 Dreams Come True

This paper presents *Soft Interfaces*, a module system with typed interfaces. Soft Interfaces fulfills the stated demands. It uses the type language and inference engine of Andrew Wright's Soft Scheme [21]. Our main contribution is to have integrated a module system with type inference for a dynamically typed language including an algorithm to infer the implementation type of abstract types. Our prototype implementation of Soft Interfaces runs within Scheme 48 [10], an R^5RS-compliant Scheme implementation.

The remainder of this paper is organized as follows: Section 2 reviews standard type-theoretical notions relating to the treatment of modules, most importantly abstract types and polymorphism. We derive an algorithm to satisfy the four demands stated above. Section 3 describes the actual implementation of Soft Interfaces. Section 4 deals with possible extensions, Section 5 reviews approaches related to ours. The paper ends with our conclusion in Section 6.

2 Type-checking for Soft Interfaces

Our intention of introducing modules is to establish a boundary at which we perform type checks. More precisely there are two separate type checking tasks:

1. The module's bindings must obey their declared interfaces.
2. The exported names must be used only in the specified way.

Although the general method is certainly standard, there are a few subtle points to be clarified about our special situation. Most notably, there does not seem to be a broad consensus in the literature on the procedure to infer the implementation of a module's abstract type, which we allow to be omitted in the source file.

Section 2.1 reviews the type system of Wright's Soft Scheme [21], which features union and recursive types and strikes a balance between expressiveness and simplicity. Section 2.2 gathers the type-theoretic tools necessary to derive a type checker for the module boundary in Section 2.3. This derivation can be carried out in a formal manner from well-established axioms and thus provides strong support, if not a proof, of our implementation's correctness. Finally, Section 2.4 outlines how our approach can be adapted to type systems other than Soft Scheme.

2.1 Soft Scheme

The type system of Soft Scheme [21] implements type inference in a theory with union types, recursive types and (first-order) polymorphism. The inference algorithm makes use of a special representation of union types with *flag variables* and *type variables*, in order to infer precise types efficiently [21, sec 2.3]. The idea is to extend the standard unification algorithm to decide the subtyping relation on union types: The type variable t allows an inferred union $\sigma \cup t$ to be extended (by substitution), while a flag variable attached to every element of the union can be set to $-$, effectively excluding the element and inhibiting a later re-introduction. Unification requires that the pairs of subexpressions to be unified are uniquely determined, which motivates the definition of *tidy types*: Each type constructor may appear at most once within a union and type variables must have a consistent range, i.e. for every type variable, the unions it is contained in must include the same type constructors. The latter restriction makes substitutons preserve tidiness.

Tidy types also include recursive types, conventionally introduced by a special construction $\mu t.\sigma$ where the scope of μ extends as far to the right as possible. For type inference with recursive types, Soft Scheme follows the approach of Amadio/Cardelli [2] and represents types internally as cyclic graphs (or equivalently infinite regular trees), which serves as a basis for deciding the subtype relation.

The syntax of the type expressions in Soft Scheme [21, sec. 2.3] is given by the following grammar, where $(\kappa^f \vec{\sigma})$ is a base type with constructor κ with flag variable $f \in \{+,-\}$ and argument vector σ whose length matches the arity of the constructor

$$\begin{array}{rcl} \sigma & = & \kappa_1^{f_1}\vec{\sigma}_1 \cup \cdots \cup \kappa_n^{f_n}\vec{\sigma}_n \cup (t \mid \emptyset) \\ & \mid & \mu t.\sigma \end{array} \qquad (1)$$

Here t is a type variable and $\mu t.\sigma$ binds t in σ. The notation $\vec{X}$ is denotes a vector $X_1, \ldots, X_n$.

As an example, the type of proper lists over type t combines the shown constructions : $\mu s.\mathtt{nil}^+ \cup (\mathtt{cons}^+\ t\ s) \cup \emptyset$.

We shall use the judgment

$$\Gamma \vdash_{\mathrm{ST}} M : \sigma$$

to express that Soft Scheme's type inference assigns type σ to expression M under type assumptions Γ. Note that according to (1) σ may contain free type variables. As is customary for formal derivations, we assume that the results of occurring unifications have already been applied to σ.

2.2 Type-theoretic Tools

We have argued in Section 1 that a module system should provide both abstract data types and polymorphic functions. The general relationship and interaction between these features has been studied widely in the literature [4, 15]. We briefly review the basic constructions and give the connection to our type checker.

Extending the Type Language We use ρ and τ to denote extended type expressions with record types, universal and existential quantification according to the following grammar, where σ is a Soft Scheme type as defined by (1).

$$\rho, \tau ::= \sigma \mid \forall t.\rho \mid \exists t.\rho \mid \{l_i : \rho_i\}_{i=0}^n \qquad (2)$$

Here $\exists t.\rho$ and $\forall t.\rho$ bind t in ρ. In this section, we shall use the word "type" both for the expressions σ and ρ, the distinction will be clear from the context. We will write $\mathbf{Q}\, t_1, \ldots, t_n.\rho$ as an abbreviation for $\mathbf{Q}\, t_1. \cdots \mathbf{Q}\, t_n.\rho$ where convenient; $\mathbf{Q}$ is $\forall$ or $\exists$.

The introduction and elimination rules for the constructs in ρ will be given in the subsequent paragraphs in terms of a judgment

$$\Gamma \vdash M : \rho.$$

Record Types The typing of labeled records has been studied extensively in connection with object-oriented programming. A comprehensive development has been given by Cardelli and Mitchell [3]. For our purposes, the following formulation taken from Cardelli and Wegner's earlier presentation [4] suffices. A record $\{l_i = d_i\}_{i=1}^n$ is interpreted as a mapping from names l_i to values d_i. Its type $\{l_i : \rho_i\}_{i=1}^n$ accordingly accumulates the types of the contained values. The introduction rule for $\{\,\}$ types is straightforward:

$$(\{\,\}\text{-intro})\ \frac{\Gamma \vdash d_i : \rho_i \text{ for } i = 1, \ldots, n}{\Gamma \vdash \{l_i = d_i\}_{i=1}^n : \{l_i : \rho_i\}_{i=1}^n}$$

Elimination is syntactically marked by an access to a member:

$$(\{\,\}\text{-elim})\ \frac{\Gamma \vdash R : \{l_i : \rho_i\}_{i=1}^n}{\Gamma \vdash R.l_i : \rho_i} \quad i \in \{1, \ldots, n\}$$

Existential Types The assertion $M : \exists t.\rho$ hides some of the inferred information about M's type by leaving subexpressions of ρ unspecified. Therefore it is natural to regard t as an abstract type in a "module" M. The proof of the assertion is constructive: We have to provide a type τ such that $\Gamma \vdash M : [\tau/t]\rho$ can be deduced. Cardelli and Wegner [4] use an operation **pack** to construct values of $\exists$ types according to the following rule

$$(\exists\text{-intro})\ \frac{\Gamma \vdash M : [\tau/t]\rho}{\Gamma \vdash \mathbf{pack}[t = \tau \ \mathbf{in}\ \rho]M : \exists t.\rho}$$

When using a module, nothing can be assumed about the abstract types, yet their names can be used in declarations of local variables. The **open** rule provides a handle for the abstract type without revealing the hidden information:

$$(\exists\text{-elim})\ \frac{\Gamma \vdash M : \exists t.\rho \quad \Gamma, m : [s/t]\rho \vdash N : \tau}{\Gamma \vdash \mathbf{open}\ M\ \mathbf{as}\ m[s]\ \mathbf{in}\ N\ :\ \tau} \quad \begin{array}{l} s \text{ not free} \\ \text{in } \tau, \Gamma \end{array}$$

The proviso about s can be read as: "s is treated locally as a name distinct from all others."

Universal Quantification Parametric polymorphism is captured by allowing a quantifier $\forall$ in the type language. Essentially two choices for its introduction rule can be made: Either a type expression can be generalized over its free variables at any point in the deduction [5] or this abstraction occurs at syntactically designated points (i.e. Λ-abstraction over types [16, 14] or *let*-bindings [12]). Since we allow polymorphism at module level and definition level, the second approach fits our needs. For ease of formulation the introduction rule nevertheless appears in the style of Cardelli and Wegner [5]:

$$(\forall\text{-intro}_1)\ \frac{\Gamma \vdash M : \rho}{\Gamma \vdash M : \forall t.\rho} \quad \text{where } t \text{ is not free in } \Gamma$$

The condition on t states that no assumptions are to be made about t in the deduction of the premise. This is particularly important when reading the rule in the spirit of a type *inference* algorithm (as opposed to a type *checking* application as is formulated here): Although t is a type variable, it is not available for substitution during unification in the derivation of $\Gamma \vdash M : \rho$. We can make this explicit by replacing it with an otherwise unused type constructor in the premise.

$$(\forall\text{-intro})\ \frac{\Gamma \vdash M : [\kappa/t]\rho}{\Gamma \vdash M : \forall t.\rho} \quad \kappa \text{ a fresh type constructor}$$

The standard elimination rule for $\forall$ allows us to replace the polymorphic variable by any type:

$$(\forall\text{-elim})\ \frac{\Gamma \vdash M : \forall t.\rho}{\Gamma \vdash M : [\tau/t]\rho}$$

Subtyping The type discipline of Soft Scheme embodied by $\vdash_{\mathrm{ST}}$ allows a restricted form of subtyping on union types by considering $\sigma_1 \leq \sigma_1 \cup \sigma_2$. Subtyping can be integrated smoothly with the type constructions reviewed so far by allowing the standard inference rule [13]:

$$(\text{subsumption})\ \frac{\Gamma \vdash M : \sigma' \quad \sigma' \leq \sigma}{\Gamma \vdash M : \sigma}$$

Typing of Modules and Scheme expressions We follow [15, sec. 3.4] in the description of a module M with exported names $l_1, \ldots, l_m$ of declared types $\sigma_1, \ldots, \sigma_m$. M may include parametric polymorphism at the level of single members, the exports are grouped into a record and an abstract type, possibly with type parameters, can be provided. The type structure of a module is therefore as follows:

$$\begin{array}{rll} \forall t_1, \ldots, t_n \exists A\{\ l_1: & \forall s_{11}, \ldots, s_{1k_1}\sigma_1 & \\ \vdots & & \\ l_m: & \forall s_{m1}, \ldots, s_{mk_m}\sigma_m\ \} & \end{array} \tag{3}$$

We have treated the type inference on Scheme expressions by Soft Scheme and have extended the type language suitably to encode modules. It remains to connect the two judgments $\Gamma \vdash_{\mathrm{ST}} M : \sigma$ and $\Gamma \vdash M : \rho$ to reflect the extension of the type languages:

$$(\vdash_{\mathrm{ST}} - \vdash)\ \frac{\Gamma \vdash_{\mathrm{ST}} M : \sigma}{\Gamma \vdash M : \sigma}$$

2.3 Deriving the Type Checker

We now turn to the two type checking tasks presented in the introduction to this section, namely checking of the binding's types against the interface types and the type-correct use of imported names. For practical utility it is important that these tasks can be carried out independently. We shall see that this is the case for our approach and derive the corresponding procedures from the general type-theoretic notions reviewed in Section 2.2. The main result of this section is the algorithm $\mathcal{S}$. It checks the module's bindings against the expected interface, inferring implementation types of possible abstract types along the way.

Derivation of Algorithm $\mathcal{S}$ In order to assign the type shown in (3) to a module M we have to introduce the quantifiers according to the rules shown in Section 2.2. Figure 1 shows the details of the deduction. The leaf of the deduction tree specifies the task of the checking algorithm if we are given the implementation type τ of abstract type A. We have to apply the substitution

$$\phi := [S_{i1}, \ldots, S_{ik_i}/s_{i1}, \ldots, s_{ik_i}][\tau/A][T_1, \ldots, T_n/t_1, \ldots, t_n] \tag{4}$$

to the interface types σ_i and then check that the bindings' inferred types match.

It remains to infer an abstract type's implementation type. A closer inspection of the leaf judgment of the deduction shown in Figure 1 suggests the way to proceed: An application of the rule (subsumption) to each of the m goals yields new goals

$$\Gamma \vdash_{\mathrm{ST}} d_i : \sigma_i' \quad i = 1, \ldots, m \tag{5}$$

together with a set of subtyping relations

$$\{\sigma_i' \leq \phi\sigma_i\}_{i=1}^{m} \tag{6}$$

The σ_i' are best read as the types inferred for the bindings by Soft Scheme's type checker. Then this deduction step explains directly the well-known relation: "The binding's type must be a subtype of the interface's type". Furthermore, the inequalities (6) can be read as constraints on the implementation type τ for abstract type A: The substitution $[\tau/A]$ within ϕ must be chosen such that all inequalities are satisfied.

Sketch of the algorithm $\mathcal{S}$ Algorithmically, we have to start at the leaves of the deduction tree. Therefore, we apply Wright's type inference procedure for $\vdash_{\mathrm{ST}}$ to each of the bindings d_i and obtain their types σ_i'.[2]

In a second step, we have to solve the subtype relations from (6). Towards that end, we apply a substitution

$$\phi' := [S_{i1}, \ldots, S_{ik_i}/s_{i1}, \ldots, s_{ik_i}][\underline{\tau}/A][T_1, \ldots, T_n/t_1, \ldots, t_n] \tag{7}$$

to the interface types σ_i, where the special variable $\underline{\tau}$ is marked unavailable for unification. We can directly employ Amadio and Cardelli's trail algorithm for subtyping recursive types [2] to decide the subtyping questions $\{\sigma_i' \leq \phi'\sigma_i\}$; only a slight adaption is needed to handle Soft Scheme's union types. The procedure recursively decomposes types, until it meets base types or variables. In case the variable $\underline{\tau}$ is compared against some $\bar{\sigma}_j$, the procedure does not directly apply a substitution but rather records this inequality, thereby gathering "propositions" for the intended implementation type:

$$\{\underline{\tau} \leq \bar{\sigma}_j\}_{j=1}^{I'-1} \cup \{\bar{\sigma}_j \leq \underline{\tau}\}_{j=I'}^{I} \tag{8}$$

As the third step, we compute a substitution $[\tau/\underline{\tau}]$ satisfying the inequalities (8). Since in the type theory of $\vdash_{\mathrm{ST}}$ the only non-trivial relations in $\leq$ are between (tidy) union types, system of inequalities (8) has a most specific solution, which if it exists can be found considering group $I', \ldots, I$ alone: We can take $\tau := \bigcup_{j=I'}^{I} \bar{\sigma}_j$, where the operation $\bigcup$ attempts to build τ as a tidy union type with $\bar{\sigma}_j \leq \tau$ and reports a type error if this is not possible. That type error signals that the programmer has made inconsistent assumptions on the implementation type. Finally, we have to check group $I, \ldots, I'-1$ for satisfaction under the replacement $[\tau/\underline{\tau}]$.

Use of imported names Loading a module involves opening its $\exists$ type and instantiating polymorphic types during elimination of $\forall$-quantifiers, both according to Section 2.2.

Algorithmically, polymorphism can be treated the usual way: The type expressions contain variables available for substitution during unification. The existential quantification of a module is eliminated by considering its abstract type as a new type name distinct from all others and allowing references to the packaged operations, in our case through member selection on records.

The actual implementation is done by extending the set of assumptions Γ in the judgments $\Gamma \vdash_{\mathrm{ST}} M : \sigma$ passed on to Soft Scheme. Technically, this is achieved by inserting the assumptions into the current environment, replacing $\forall$-bound variables with free type variables.

[2] Strictly speaking, the σ_i' are substitution instances of the inferred types: The formal derivation assumes that ($\forall$-elim) rules have been applied as needed in the deduction of $\Gamma \vdash_{\mathrm{ST}} d_i : \sigma_i'$. This detail is handled by unification as usual.

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash_{\mathrm{ST}} d_i : [S_{i1},\ldots,S_{ik_i}/s_{i1},\ldots,s_{ik_i}][\tau/A][T_1,\ldots,T_n/t_1,\ldots,t_n]\sigma_i \quad \text{for } i=1,\ldots,m}{\Gamma \vdash d_i : [S_{i1},\ldots,S_{ik_i}/s_{i1},\ldots,s_{ik_i}][\tau/A][T_1,\ldots,T_n/t_1,\ldots,t_n]\sigma_i \quad \text{for } i=1,\ldots,m}}{\Gamma \vdash d_i : [\tau/A][T_1,\ldots,T_n/t_1,\ldots,t_n]\forall s_{i1},\ldots,s_{ik_i}\sigma_i \quad \text{for } i=1,\ldots,m}}{\Gamma \vdash M : [\tau/A][T_1,\ldots,T_n/t_1,\ldots,t_n]\{l_i : \forall s_{i1},\ldots,s_{ik_i}\sigma_i\}_{i=1}^m}}{\Gamma \vdash M : [T_1,\ldots,T_n/t_1,\ldots,t_n]\exists A\{l_i : \forall s_{i1},\ldots,s_{ik_i}\sigma_i\}_{i=1}^m}}{\Gamma \vdash M : \forall t_1,\ldots,t_n\exists A\{l_i : \forall s_{i1},\ldots,s_{ik_i}\sigma_i\}_{i=1}^m}$$

Figure 1: Derivation of a module's type

2.4 Beyond Soft Scheme

We have presented a procedure to integrate Soft Scheme's type inference algorithm with the checking of typed interfaces. Our presentation also reveals the relation between these two typing issues: In the check of a module's implementation against its interface as shown in Figure 1, the inference rule ($\vdash_{\mathrm{ST}}$-$\vdash$) separates the introduction of the module's type (cf. 2.2– (3)) from the type inference $\Gamma \vdash_{\mathrm{ST}} d_i : \phi\sigma_i$, $i = 1,\ldots,n$ on bindings.

In case type inference on the bindings features a subtyping relation, rule (subsumption) increases the programmer's flexibility when bindings are checked against types declared in the interface. The rule (subsumption) also splits our algorithm $\mathcal{S}$ into the steps of type inference and of solving a set of inequalities. We obtain constraints on the implementation types for abstract types by analyzing these inequalities suitably. This part of our algorithm does not impose any restrictions on the subtyping relation, apart from the existence of least upper bounds, but these are already required for principal types. Therefore we conjecture that our algorithm is applicable to type disciplines other than Soft Scheme as well.

3 Soft Interfaces

We now present a description of Soft Interfaces, a module system for Scheme with support for explicit type declarations. Soft Interfaces is built within Scheme 48. It uses the type system and type inference system of Soft Scheme. The next sections specify the syntax and semantics of Soft Interfaces and describe the implementation.

3.1 Syntax of the Module Language

The syntax of the module language of Soft Interfaces is close to Scheme 48's. The keyword `define-interface` declares an interface which may be referred to by *name*, the body contains one `soft-export` declaration:

```
(define-interface name
  (soft-export clause ...))
```

A *clause* is can be a pair of name and type:

clause $\rightarrow$ (*identifier type*)

This states that *identifier* is bound to a value with type *type*. The notation for types is derived from Definition (1):

type	$\rightarrow$	`(+` *type* `...)`	
	$\mid$	`(mu` *identifier type*`)`	
	$\mid$	`(`*k type* `...)`	*k* a type constructor
	$\mid$	*typevar*	
typevar	$\rightarrow$	`'`*identifier*	

Furthermore, a *clause* may contain the definition of an abstract type which consists of a name and a listing of parameters for the type:

clause $\rightarrow$ ...
$\mid$ `(typedef (`*identifier typevar* `...))`

As an example, consider the interface of a small library for extensible, homogeneous vectors:

```
(define-interface date-interface
  (soft-export
    (abstract-type (vector 'X1))
    (make-vector (num 'X1 -> (vector 'X1)))
    (extend-vector ('X1 (vector 'X1) -> (vector 'X1)))
    (vector-ref ((vector 'X1) num -> 'X1))))
```

Modules are called *structures* in Scheme 48 and are introduced by the `define-structure` form:

```
(define-structure structure-name interface-name
   (open import-name ...)
   (files file-name ...))
```

This defines a structure with the name *structure-name*. The identifier *interface-name* refers to an interface which determines the exported identifiers of the structure. The *import-names* refer to other structures, which are made visible within the structure but must not import *structure-name* directly or indirectly. *File-names* list the files which contain the source code for the structure.

3.2 Semantics of the Module Language

We now give an informal description of the semantics of Soft Interfaces. It comes into place when the forms of the module language are evaluated. Interface declarations evaluate to a mapping from names to types. A structure is represented as a triple:

- A reference to the interface
- A list of imported structures
- Either a reference to the source file or a *package* if the structure is already loaded.

A package is a mapping from names to binding information. The binding information consists of internal data for the compiler along with the type inferred by the type checker.

Structures support two operations:

`Load` This command recursively ensures that all structures in the import list are loaded and calls the byte-compiler to evaluate the file. The compiler returns a package. At this point, types in the package have to be checked to meet the type declarations in the interface. To that end we call algorithm $\mathcal{S}$ with the declared types from the interface and the inferred type from the binding.

If they do not meet, an error is signaled and the structure is not loaded. Otherwise, the package is stored in the third field of the structure.

In any case, subsequent `load` commands for the same structure will be ignored.

`Lookup` *`name`* Here, first the interface of the structure is queried for *name*. In case *name* is not contained in the interface, `lookup` immediately returns false. Otherwise, the interfaces provides a type. Afterwards, the structure's package is searched for a binding for *name*. If this fails `lookup` is called on the imported structures. This ends up with a binding information which `lookup` returns in a pair along with the type from the interface.

`Lookup` can only be called if the structure is loaded.

3.3 Combination of Soft Scheme with the Module System

This section describes the intregration of the Soft Scheme system into the Scheme 48 byte-code compiler. For a better understanding, we give a brief overview of the compiler stages of Scheme 48 without Soft Interfaces:

- At the request of the user the compiler loads a file containing the module declaration. This declaration is then processed by the interpreter of the module language which returns a structure.
- The user invokes the `load` operation on that structure.
- The standard reader loads the source files referenced in the structure and generates s-expressions from the source code.
- The parser generates an abstract syntax tree in which the nodes are represented by a record holding the type of the node, the sub-nodes and a property list to store arbitrary data such as binding information.
- The compiler invokes a number of source-to-source optimizers on the abstract syntax tree. These optimizers either perform simple transformations on the tree or add hints for the compiler to the property list of various node. Some of the optimizers are only applied if the module definition contains an appropriate request.
- The back-end generates the byte-code by traversing the abstract syntax tree and reading annotations inserted by the optimizers.
- The linker builds binding information that map names to memory locations.

The actual implementation of algorithm $\mathcal{S}$ as specified in Section 2.3 takes place at two distinct points in the above sequence. The type inference engine of Soft Scheme is presented to Scheme 48 as an additional optimizer pass during compilation. The actual subtyping judgment between the declared and inferred types takes place during the `load`-operation of the structure.

As mentioned above, implementing an optimizer means that we have to provide a procedure which is called with the abstract syntax tree and the structure as arguments and returns a modified abstract syntax tree. To incorporate Soft Scheme, we had to adapt its front end to match Scheme 48's abstract syntax tree. Soft Scheme accesses the type information of imported identifiers by calling the `lookup` operation of the structure containing that identifier. After type inference, Soft Scheme introduces checks for primitives and adds type information for expressions as new kinds of node. We chose not to store them in the property lists to retain as much of Soft Scheme's structure as possible. The rest of the compiler currently ignores these new nodes, but the check nodes could be used during code generation. To provide access to the types of identifiers from other structures, the optimizer finally stores the types in the binding information of the identifiers. At this place, type checking is done and we can return the abstract syntax tree to the compiler which proceeds with other optimizers.

4 Extensions

Several extensions to Soft Interfaces are conceivable. This section will propose type macros, the declaration of predicates, checkable types and run-time check elimination.

4.1 Type Macros

Typed interfaces provide a good source for documentation. With abstract types it is possible to enrich this documentation with meaningful names for arguments and results of procedures and hide the implementation of a type at the same time. The resulting incompatibility with other types is sometimes not desirable. Consider for example Scsh's [19] procedure `dup->fdes`. It takes a port or a file descriptor as argument, performs the system call `dup` to duplicate the file descriptor and returns the new file descriptor. Ports are declared as abstract type `port`. To enable communication with other programs file descriptors are represented as integers in Scsh and cannot be made abstract. A declaration for `dup->fdes` using Soft Interfaces would therefore be written like this:

```
(dup->fdes ((+ port num) -> num))
```

Here the information that the `num`s stand for file descriptors is not apparent. We remedy this by providing type macros via the `defalias` form in the clauses of interface declarations:

$$\begin{array}{lcl} \textit{clause} & \rightarrow & \ldots \\ & | & (\texttt{defalias}\ \textit{name}\ \textit{type}) \end{array}$$

Type macros are expanded before the type declarations are handed out to Soft Scheme. The declaration of `dup->fdes` now looks like this:

```
(defalias fd num)
(dup->fdes ((+ port fd) -> fd))
```

From this type the user can conclude what `dup->fdes` does but apply it to numbers as before.

The combination of file descriptors and ports is quite common in Scsh, so a macro `fd/port` is possible convenient:

```
(defalias fd/port (+ fd port))
```

As nested unions are prohibited in Soft Scheme's presentation types `fd/port` cannot be used within another union. However, a context sensitive macro expander can flatten out unions and thus effectively permit nested unions.

4.2 Down with Complex Types

Static type systems are often faced with the criticism that the inferred types are too complicated and consequently inappropriate for daily use. As we stated earlier, the restriction to types of exported procedures eases much of the complexity. Two new types `top` and `bottom` turn out to be useful in type declarations. Their purpose lies in the subtype relation to the rest of the types: all other types are subtypes of `top` and `bottom` is a subtype of every other type.

With `true` and `false` the programmer has the possibility to decrease complexity by substituting `top` for excessive types in result position of procedures and likewise `bottom` in argument positions. Of course this entails a sacrifice of accuracy for simplicity.

A new case in the clauses makes it even possible to omit type declarations completely:

$$\begin{array}{rcl} \textit{clause} & \rightarrow & \ldots \\ & | & \textit{name} \end{array}$$

This rule is merely an abbreviation of `(name top)`.

4.3 Predicates

Programmers use predicates mostly in the test expressions of conditionals. In practice this means that the type of an expression decides which code will be evaluated next and that the chosen code can rely on the result of the predicate test. It is therefore essential for good accuracy to treat tests with predicates specially. Soft Scheme includes sophisticated inference for ML-like `match` expressions and converts Scheme's `if` expressions involving predicates into these `match` expressions. With these conversions programs that use `if` benefit from the type inference rules for `match`, too.

As predicates provide useful information for accuracy, they should not be limited to the standard set provided by R^5RS but extensible to user defined data types and to abstract types. To accomplish this, a new case for clauses is added to Soft Interfaces:

$$\begin{array}{rcl} \textit{clause} & \rightarrow & \ldots \\ & | & (\texttt{?}\ \textit{name type}) \end{array}$$

This declares *name* to be a predicate for type *type*. Whenever such a predicate is used as test of an `if` expression, the system converts the `if` to a corresponding `match`.

Some care by the user is required, as the system can only check the binding to have type `('X -> bool)`, whether it returns true only for *type* is a semantic issue and in general undecidable.

4.4 Checkable Types

Consider Scsh's procedure `read-symlink` with type

```
(string -> string)
```

It is used to dereference the symbolic link specified in the first argument as a filename. `Read-symlink` is implemented via a system call provided by the C-library, hence the filename has to be converted to a `char*`. At this point, an exception is thrown if `read-symlink` was called with a non-string argument. The connection between the exception and the type error may not be not obvious to the user.

In order to provide meaningful error messages, the implementors of Scsh add code to check the types of arguments directly at the beginning of exported procedures. This is a tedious task and it means in fact to implement the type specification of the procedures.

Soft Interfaces can be of aid to the implementor here as it knows the type of exported procedures. Calls with arguments matching these types can never go wrong. However, if the type check fails the system can either reject the program or check the arguments at run-time. The first alternative would make the type system a part of the language definition rather than a debugging tool so only the second idea remains.

However, in the presence of higher-order functions, polymorphism and recursive types, checking the arguments of procedures at run-time is not possible in general, because checking whether a value belongs to such a type is undecidable. A simple example is the procedure

```
(map (('X1 -> 'X2) (list 'X1) -> (list 'X2)))
```

It is not possible to check that the type of the first argument's domain is the same as the type of the list's contents.

For a useful class of ground types, it is always possible to construct *analyzers*, nevertheless. An analyzer behaves much like a predicate but also recursively checks a value's sub-components by analyzers supplied for each of the type parameters. With these we can define the class of *checkable types* ω:

$$\begin{array}{rcl} \omega & = & (+\ \omega\ \ldots) \\ & | & \{\kappa\ a_1 \ldots a_n \mid \exists \text{ analyzer for } \kappa \wedge a_i \in \omega\} \end{array}$$

Analyzers can be declared in the interface similar to predicates in the previous section. The reason for excluding procedures from checkable types is that checking the result type of a procedure would break proper tail-recursion.

If the declared type of an argument is checkable, the compiler can insert checks with the corresponding analyzer either at call-site or at the beginning of the procedure. For `read-symlink` both alternatives will work and free the implementor from adding a test with `string?` manually.

4.5 Revisiting Run-time Checks

One of the main features of static type systems is the elimination of run-time checks for primitives. With Soft Interfaces this optimization is even possible in combination with modules. The key idea is to provide two versions of each procedure: One is equipped with full run-time checks at primitives and can be called without further provisions. The second contains only those checks that are necessary in case the interface is obeyed in the call. While the first version enables us to never reject a program due to a type error, calls to the second version rely on the basic observation: If a procedure is called according to its exported type, nothing can go wrong.

For the elimination of run-time checks we cannot adopt the strategy of the underlying Soft Scheme system: For every procedure, that system inserts checks only if any of the calls to the procedure requires this. This approach will not work in the context of modules as not all calls to exported procedures are known during the compilation of a module.

If the types of arguments and result of a procedure are all checkable, we can use the unchecked version in any case, since all run-time checks can be performed at the call-site.

The method described above embodies the first strategy for elimination of run-time checks which supports separate compilation. The implementation and measurement of it is part of our ongoing research. Especially, we have to investigate whether enough run-time checks vanish to justify code duplication.

5 Related Work

We know of two other Scheme systems with support for typed interfaces: Scheme 48 and Bigloo [17]. Both support annotations with basic types and *top*. Further on, Bigloo's type language encompasses the types of user defined object oriented classes. Both of them lack support for parametric polymorphism, recursive types and unions and hence provide only limited accuracy.

Outside the domain of type systems, implementors use flow analysis to determine whether all arguments to primitives are of correct type [18, 7, 9]. However this approach has limited use if only parts of the program are available for analysis. If a procedure is never called at all, no values will flow into this procedure causing no analysis to take place. The lack of calls is the typical setup in the case of libraries and embedded domain specific languages. The graphical debugger MrSpidey [6], likewise based on flow analysis, makes it possible to check calls with abstract values and therefore to simulate tests on an infinite number of arguments. If we add the call

```
(merge-sorted (type: (list str))
              (type: (list str))
              string<)
```

and hand the program from the introduction out to MrSpidey, it would report a check at the primitive `<`, which reveals our bug. However, MrSpidey does not support parametric polymorphism within these artificial calls which is exactly the lack of communication from the programmer to the type system we pointed out in the introduction.

6 Conclusion

In the introduction we have singled out four specific demands to aid static debugging of Scheme programs: Type declarations, a powerful type language, separate checking for parts of programs and abstract types. Modules with typed interfaces do meet these demands. Moreover, they allow the declarations to be kept apart from the definitions, leaving the source code unaltered.

We have derived the required checking algorithm $\mathcal{S}$ for typed-interfaces from standard type-theoretical notions in Section 2. In particular we were able to give a procedure that determines the implementation type for an abstract type from the implementation of its operations.

Our implementation of typed interfaces called Soft Interfaces combines the Soft Scheme system by Andrew Wright with the module system of Scheme 48. The link between them is our checking algorithm $\mathcal{S}$. Soft Interfaces is able to solve the problems exemplified in the introduction.

As extensions we have proposed type macros, the notion of checkable types and their application to the elimination of run-time checks.

Acknowledgments We would like to thank Michael Sperber for invaluable comments on previous drafts of this paper. The authors also thank the anonymous referees for their helpful and constructive comments.

References

[1] A. Aiken, E. L. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Proc. 21st Annual ACM Symposium on Principles of Programming Languages*, Portland, OG, Jan. 1994. ACM Press.

[2] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[3] L. Cardelli and J. C. Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1(1):3–48, 1991. also available as DEC SRC Research Report 48.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–522, Dec. 1985.

[5] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–520, December 1985.

[6] C. Flanagan. MrSpidey. `http://www.cs.rice.edu/CS/PLT/packages/mrspidey/index.html`, Dec. 2000.

[7] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. *ACM SIGPLAN Notices*, 31(5):23–32, May 1996. Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).

[8] F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In S. Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, La Jolla, CA, June 1995. ACM Press, New York.

[9] S. Jagannathan and A. Wright. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, 20(1):166–207, Jan. 1998.

[10] R. A. Kelsey and J. A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[11] B. Liskov. *CLU reference manual*. Springer Verlag, 1981. LNCS 114.

[12] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[13] J. C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, July 1991.

[14] J. C. Mitchell and G. D. Plotkin. Abstract Types Have Existential Type. *ACM Transactions on Programming Languages and Systems*, 10(8):470–502, July 1988.

[15] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[16] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Proceedings: Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–423. Springer-Verlag, 1974.

[17] M. Serrano. Bigloo User's Manual for Bigloo 2.3a. `http://kaolin.unice.fr/~serrano/bigloo/`, June 2001.

[18] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1991. Also technical report CMU-CS-91-145.

[19] O. Shivers. A Scheme Shell. Technical Report MIT-LCS//MIT/LCS/TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, Apr. 1994.

[20] A. K. Wright. *Practical Soft typing*. PhD thesis, Rice University, Houston, Texas, 1994.

[21] A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1):87–152, January 1997.