JFP **17** (2): 207–214, 2007. © 2007 Cambridge University Press
doi:10.1017/S0956796806006204 First published online 4 January 2007 Printed in the United Kingdom

Type sharing constraints and undecidability

PHILIPPE NARBEL

LaBRI, University of Bordeaux 1, 351, Cours de la Libération, 33405 Talence, France
(*e-mail:* narbel@labri.fr)

Abstract

Let $\mathscr{A}$ be a set of modules and parameterized modules including type sharing constraint specifications. We prove that determining the set of the effective modules described by $\mathscr{A}$ is undecidable. As a consequence, type sharing constraints are proved to be not always avoidable by constructive transformations.

1 Introduction

When composing strongly typed components, explicit coherence type constraints, i.e. *type sharing constraints*, are sometimes necessary to ensure static type analysis (MacQueen, 1986; Harper *et al.*, 1987; Harper & Mitchell, 1993). Incrementally constructed systems can then be safely implemented from compatible components. However, type sharing constraints may also make difficult the understanding of a software architecture that includes parameterized components like ML functors. If $\mathscr{C}$ is the set of the components which can be obtained from such an architecture by instantiating its parameterized components, one may indeed wonder whether $\mathscr{C}$ fulfills some programmer's needs. For instance, one could ask whether $\mathscr{C}$ contains components satisfying a given signature, or whether $\mathscr{C}$ contains components with coherent types that can be composed together. In the general case, we show that these questions are undecidable. The proof of this result does not rely on a particular ML-like type semantics, but only on the use of simple type sharing constraints. As such, this result can also be related to general software reuse frameworks like *generic*, *parameterized* or *generative programming* (Goguen, 1984; Krueger, 1992; Czarnecki & Eisenecker, 2000; Gibbons & Jeuring, 2003).

A consequence of this undecidability result is that explicit type sharing constraints cannot always be eliminated by constructive transformations applied to the components (a question already addressed (MacQueen, 1986; Jones, 1996; Harper, 2002; Harper & Pierce, 2005)). Type sharing specifications are thus irreducible, while they are known to induce many complications in the language semantics (Milner *et al.*, 1987).

2 Basics

We first recall some basic facts about ML module systems. *Signatures* are types and also interfaces for the implemented modular components. For instance, here is the

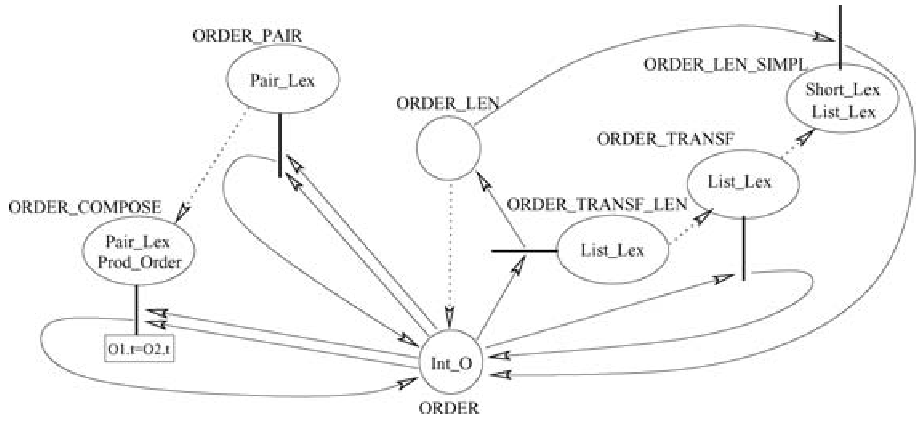

Fig. 1. A representation of the set $\mathscr{A}$ of modules, functors, and signatures as given in the text. Each vertex corresponds to a signature: simple vertices for module signatures, and vertices with "handles" for functor signatures. Each parameter signature is linked to the handle of its functor, which is in turn linked to the result signature. Each vertex-signature contains all of the components in $\mathscr{A}$ which satisfy it. The dotted arcs denote compatibility (subtype) relationships. The labeled handle of *ORDER_COMPOSE* (a constrained version of *ORDER_PAIR*) indicates its involved type sharing constraint. Applying the available functors yields new components which can also take their place in the vertices.

classic example about the ordered integers in SML (Paulson, 1992):

```
signature ORDER = sig                    structure Int_O : ORDER = struct
  type t                                   type t = int
  val leq : t * t -> bool                  val leq (x, y) = ... (* some order *)
end;                                     end;
```

Functors are module functions – parameterized modules – with typed parameters. For example, here is a functor generating new ordered sets of pairs out of two ordered sets:

```
functor Pair_Lex (O1: ORDER) (O2: ORDER) = struct
  type t = O1.t * O2.t
  fun leq ((x1, y1), (x2, y2)) = ...
     (* lexicographic order over pairs using O1.leq and O2.leq *)
end;
```

A generic ML *architecture* is a collection of modules, functors and signatures. Such an architecture may evolve by applying functor expressions. Full libraries can be designed following this technique (Biagnioni *et al.*, 2001; Harper, 2002; Narbel, 2005). For instance, developing further the above example, we give here a sketch of an "order machine" which generates many different ordered sets from other ones. The following functor `List_Lex` generates *lexicographic* orders over lists of ordered data, and the functor `Short_Lex` generates *shortlex* orders over structures having a length function (see figure 1):

```
functor List_Lex (O: ORDER) = struct
  type t = O.t list
  val length = List.length
  fun leq (l1, l2) = ... (* lexicographic order using O.leq *)
end;
```

```
signature ORDER_LEN = sig
  type t
  val leq: t * t -> bool
  val length: t -> int
end;

functor Short_Lex (O: ORDER_LEN) = struct
  type t = O.t
  fun leq (l1, l2) = ... (* shortlex using O.leq and O.length *)
end;
```

Functor signatures can also be described in an explicit way in some ML dialects, e.g. OCaml and Moscow ML. For instance, here are signatures for the above functors:

```
signature ORDER_TRANSF =  functor(O : ORDER) -> ORDER;
signature ORDER_PAIR = functor(O1 : ORDER) ->
                       functor(O2 : ORDER) -> ORDER;
signature ORDER_TRANSF_LEN = functor(O : ORDER) -> ORDER_LEN;
signature ORDER_LEN_SIMPL = functor(O : ORDER_LEN) -> ORDER;
```

Sometimes functors need *type sharing constraints*, i.e. explicit type equalities to ensure type soundness of their bodies. A typical case occurs when the elements of the functor parameters are composed, necessarily imposing type coherence:

```
signature OP = sig       functor Mix (structure M1 : OP and M2 : OP
  type t                             sharing type M1.t = M2.t) = struct
  val f : t -> t           fun g x = M1.f (M2.f x)
end;                     end;
```

Type sharing constraints in ML can also be expressed by directly adding type equations to signatures. Here is a functor which generates the *product order* by composing two orders:

```
functor Prod_Order (O1: ORDER) (O2 : ORDER where type t = O1.t) =
  struct
    type t = O1.t
    fun leq (x, y) = O1.leq (x, y) andalso O2.leq (x, y)
  end;
```

This functor matches a constrained version of ORDER_PAIR, and it can also take its place in the order machine (see figure 1). The whole example illustrates the fact that architectural complexity can be obtained with few generic components.

3 The undecidability of type sharing constraints

Given an ML architecture $\mathscr{A}$ as above, what can be generated from it? For instance, let F be a functor with two parameters X_1, X_2 such that X_1 must satisfy ORDER. Then, one could ask whether $\mathscr{A}$ would be able to generate orders for X_1 that are type-coherent with the arguments of X_2. In a global setting, when type sharing constraints are involved, this kind of question is undecidable. In order to prove it in a general context, we first define a skeletal typed module language syntax derived from TypModL (Harper *et al.*, 1987; Leroy, 1996) and restricted to our needs, i.e. structures, first-order functors, signatures and generative type declarations. We add signature names, tuples of types, and n-ary functors. Let t range over type names, x over structure names, y over signature names, f over functors names:

Programs :	$m \rightarrow$	ϵ | `structure` $x = s;\ m$
		| `functor` $f\ \{(x_i : S)\}^+ = s;\ m$
		| `signature` $y = S;\ m$
Structure expressions :	$s \rightarrow$	p_s | `struct` d `end` | $f\ \{(s)\}^+$
Structure paths :	$p_s \rightarrow$	x | $p_s.x$
Structure bodies :	$d \rightarrow$	ϵ | $c;\ d$
Definitions :	$c \rightarrow$	`type` $t = T$ (type binding: non generative)
		| `datatype` t (type creation: generative)
Signature expressions :	$S \rightarrow$	y | `sig` D `end`
Signature bodies :	$D \rightarrow$	ϵ | $C;\ D$
Specifications :	$C \rightarrow$	`type` t
		| `sharing type` $p_t = p'_t$ (type constraint)
Type expressions :	$T \rightarrow$	p_t | (T)
		| $T_1\{*\ T_i\}^*$ (tuples of types)
Type paths :	$p_t \rightarrow$	t | $p_s.t$

A type definition `type` $t = T$ defines t as a synonym for the type expression T, and type generativity occurs only for `datatype` definitions. A usual "stamp-based" semantics can be applied (Milner *et al.*, 1987; Leroy, 1996).

Theorem
Let $\mathscr{A}$ be a set of signatures, modules, and parameterized modules including type sharing constraints. It is undecidable to determine whether or not a signature S_0 can be instantiated from $\mathscr{A}$. More generally, it is undecidable to determine the set of components that can be generated from $\mathscr{A}$.

Proof
We apply a reduction of the unsolvable *dominoes Post's correspondence problem.* Recall that this problem is the following: Let $D = \{D_i = (\frac{up_i}{down_i})\}_{i=1,\dots,m}$ be a set of dominoes where up_i and $down_i$ are words over some finite alphabet Σ. The question is to find a sequence of dominoes $(\frac{up_{i_1}}{down_{i_1}})(\frac{up_{i_2}}{down_{i_2}})\dots(\frac{up_{i_n}}{down_{i_n}})$, such that: $up_{i_1}up_{i_2}\dots up_{i_n} = down_{i_1}down_{i_2}\dots down_{i_n}$. The existence of such a solution is known to be undecidable (Davis & Weyuker, 1985).

We prove here that for each domino set D, there exists a collection of modules and functors for which the signature S_0 instantiation problem is solvable iff the correspondence problem with D is solvable. The basic elements of the reduction are the following: For each letter a in the domino alphabet Σ, we define a distinct type:

```
datatype a = A
```

These "type-letters" can be concatenated into words by using the type product. The word $w = s_1s_2\dots s_n$, with $s_i \in \Sigma$, is represented by:

```
type t_w = s1 * ( s2 * ... (... * sn))...)
```

We impose right associativity so that words have a unique type representation. Recall indeed that the Cartesian product of types is not associative, e.g. `(a * b) * c` $\neq$ `a * (b * c)`.

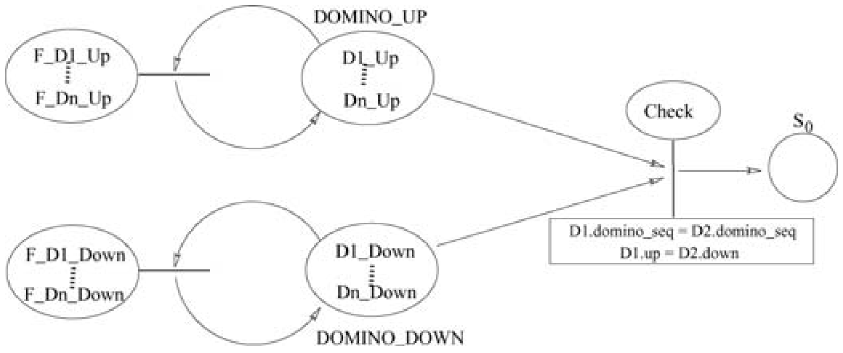

Fig. 2. A representation of the Post reduction architecture. The labeled "handle" of the `functor` signature indicates the type sharing constraints between its two parameters.

Next, these words can be concatenated together by functors. Let $w = s_1 s_2 \dots s_n$ be a word and let w' be another word denoted by a type `t` contained in some module. Then, the "concatenating functor" to obtain ww' is:

```
signature TYPE = sig type t end

functor F_w (M: TYPE) = struct
   type t = s1 * ( s2 * ... (... * (sn * M.t))...)
end
```

Now, dominoes are represented as instances of the following signatures:

```
signature DOMINO_UP = sig        signature DOMINO_DOWN = sig
  type domino_seq                  type domino_seq
  type up                          type down
end                              end
```

For each domino $D_i = (\frac{s_1 s_2 \dots s_n}{t_1 t_2 \dots t_m}) \in D$, we define a distinct type:

```
datatype d_i = Dom_i
```

We also define two modules and two concatenating functors (see figure 2):

```
structure Di_Up = struct
  type domino_seq = d_i
  type up = s1 * ( s2 * ... (... * sn)...)
end

structure Di_Down = struct
  type domino_seq = d_i
  type down = t1 * ( t2 * ... (... * tm)...)
end

functor F_Di_Up (D : DOMINO_UP) = struct
   type domino_seq = D.domino_seq * d_i
   type up = s1 * ( s2 * ... (... * (sn * D.up))...)
end

functor F_Di_Down (D : DOMINO_DOWN) = struct
   type domino_seq = D.domino_seq * d_i
   type down = t1 * ( t2 * ... (... * (tm * D.down))...)
end
```

Next, we define a particular functor over a composition of modules satisfying DOMINO_UP and DOMINO_DOWN:

```
functor Check (D1 : DOMINO_UP)
              (D2 : sig type domino_seq
                        type down
                        sharing type D1.up = down
                        sharing type D1.domino_seq = domino_seq
                    end) = struct
  ... (* satisfying S_0 *)
end
```

The signature of D2 is a constrained version of DOMINO_DOWN which makes use of elements in D1 (the parameter sequence is assumed to define nested environments).

Considering the above components as the architecture $\mathscr{A}$, what are the conditions so that Check can be applied to instantiate S_0? According to the type sharing constraints, this can happen either if some modules Di_Up and Di_Down are such that Di_Up.up = Di_Down.down, or if there are some functor application sequences D1 = F_Di_1_Up (F_Di_2_Up (... (F_Di_n_Up (Di_Up))...) and D2 = F_Di_1_Down (F_Di_2_Down (... (F_Di_n_Down (Di_Down))...) such that D1.up is equal to D2.down, that is, $\mathtt{s}_{j_1} * (\mathtt{s}_{j_2} * (\ldots * \mathtt{s}_{j_k}))\ldots) = \mathtt{t}_{h_1} * (\mathtt{t}_{h_2} * (\ldots * \mathtt{t}_{h_k}))\ldots)$. These equalities necessarily mean that $\mathtt{s}_{j_i} = \mathtt{t}_{h_i}$ for every $i = 1...k$. Moreover, the equality D1.domino_seq = D2.domino_seq ensures the synchronization of the up and down parts of the dominoes, so that the application sequence induces a well-defined domino sequence. Therefore, $Di_1Di_2...Di_nDi$ must be a solution of the Post's problem over D. Conversely, if there is a solution of the Post's problem, the above architecture emulates it by a sequence of F_Di_Up's and F_Di_Down's applications. The reduction is complete. ◇

Note that this proof can be simplified in the case that no type synchronization is required. Indeed, consider the following signature:

```
signature DOMINO_CONSTR = sig
  type up
  type down
  sharing type up = down
end
```

Dominoes can be directly instances of the signature:

```
signature DOMINO = sig
  type up
  type down
end
```

Thus, the reduction is simplified by defining for each domino $D_i = (\frac{s_1 s_2 ... s_n}{t_1 t_2 ... t_m}) \in D$:

```
structure Di = struct
   type up = s1 * ( s2 * ... (... * sn)...)
   type down = t1 * ( t2 * ... (... * tm)...)
end

functor F_Di (D : DOMINO) = struct
   type up = s1 * ( s2 * ... (... * (sn * D.up))...)
   type down = t1 * ( t2 * ... (... * (tm * D.down))...)
end
```

Here, similarly to the above proof, the reduction amounts to check the existence of some functor application sequence producing an instance of `DOMINO_CONSTR`.

Note also that the above undecidability result does not hold if type sharing constraints do not occur in $\mathscr{A} \cup S_0$. Indeed, let $\mathscr{S}$ be the set of signatures in $\mathscr{A} \cup S_0$, let $\mathscr{M}$ be the set of modules in $\mathscr{A}$, and $\mathscr{F}$ be its set of functors. Assume that for each component in $\mathscr{M} \cup \mathscr{F}$ there is some corresponding signature in $\mathscr{S}$. The set of all the components satisfying S_0 can be described by a context-free grammar (see also (Batory & O'Malley, 1992; Czarnecki & Eisenecker, 2000)): the *non-terminals* are the signatures in $\mathscr{S}$ and the *terminals* are the components in $\mathscr{M} \cup \mathscr{F}$. The *production rules* are first: $S_2 \to S_1$, with $S_1, S_2 \in \mathscr{S}$ if S_1 is compatible (a subtype) with S_2; second, $RES \to SF(S_1)(S_2)...(S_n)$ if $(S_1, S_2, \ldots, S_n) \to RES$ is a functor signature in $\mathscr{S}$; and third, $S_1 \to M_1$ if $S_1 \in \mathscr{S}$ and $M_1 \in \mathscr{M} \cup \mathscr{F}$, such that M_1 satisfies S_1. Denote by $L(S_0)$ the language defined with these rules with start symbol $S_0 \in \mathscr{S}$. One can check that this language consists exactly of all the components with signature S_0 which can be generated from $\mathscr{A}$. Now, it is well-known that $L(S_0) = \emptyset$ is decidable (see e.g., (Davis & Weyuker, 1985)).

Therefore, type sharing constraints are at the root of the above undecidability result, and they occur in a natural way when composing well-typed components. They also imply many complications in the ML semantics (Milner *et al.*, 1987). General techniques have been described to avoid them (MacQueen, 1986; Harper, 2002; Harper & Pierce, 2005): the idea is to express abstractions with regard to the components that need type sharing constraints so that type equivalences become true by construction. However, these transformations are not always possible without changing the meaning of a program:

Corollary
Let $\mathscr{A}$ be a set of signatures, modules, and parameterized modules. Then, the type sharing constraints occurring in $\mathscr{A}$ cannot always be eliminated in a constructive way.

Proof
We just have seen that the signature instantiation problem is decidable when no type sharing constraints occur. Hence, the constraints in $\mathscr{A}$ cannot always be eliminated by constructive transformations without contradicting the above theorem. $\diamond$

References

Biagioni, E., Harper, R. & Lee, P. (2001) A network protocol stack in Standard ML. *Higher Order Symbol. Comput.* **14**(4), 309–356.

Batory, D. & O'Malley, S. (1992) The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth.* **1**(4), 355–398.

Czarnecki, K. & Eisenecker, U. W. (2000) *Generative Programming*. Addison Wesley.

Davis, M. D. & Weyuker, E. J. (1985) *Computability, Complexity and Languages*. Academic Press.

Gibbons, J. & Jeuring, J. (2003) Generic Programming. *IFIP Working Conference on Generic Programming*. Kluwer Academic.

Goguen, J. (1984) Parameterized programming. *IEEE Trans. Softw. Eng.* **SE-10**(5), 528–543.

Harper, R. (2002) *Programming in Standard ML*. Carnegie Mellon University. Lecture Notes.

Harper, R. & Mitchell, J. (1993) On the type structure of Standard ML. *ACM Trans. Program. Lang. Syst.* **15**(2), 211–252.

Harper, R., Milner, R. & Tofte, M. (1987) A type discipline for program modules. *Colloquium on Functional and Logic Programming and Specifications (CFLP) on TAPSOFT '87*, pp. 308–319. Springer-Verlag.

Harper, R. & Pierce, B. C. (2005) Design considerations for ML-style module systems. In: B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pp. 293–345. MIT Press.

Jones, M. P. (1996) Using parameterized signatures to express modular structure. *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 68–78.

Krueger, C. (1992) Software reuse. *ACM Computing Surveys*, **24**, 131–183.

Leroy, X. (1996) A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, **6**(5), 667–698.

MacQueen, D. (1986) Using dependent types to express modular structure. *Proceedings 13rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 277–286.

Milner, R., Tofte, M., Harper, R. & MacQueen, D. (1987) *The Definition of Standard ML (Revised)*. MIT Press.

Narbel, Ph. (2005) *Programmation fonctionnelle, générique et objet (Une introduction avec le langage OCaml)*. Vuibert, Paris.

Paulson, L. C. (1997) *ML for the Working Programmer*. Cambridge University Press.