Reuse by program transformation*

Ralf Lämmel

CWI, P.O. Box 94079, NL-1090 GB Amsterdam ralf@cwi.nl WWW home page: http://www.cwi.nl/~ralf

Abstract

Certain adaptations, that are usually performed manually by functional programmers are formalized by program transformations in this paper. We focus on adaptations to obtain a more reusable version of a program or a version needed for a special use case. The paper provides a few examples, namely propagation of additional parameters, introduction of monadic style, and symbolic rewriting. The corresponding transformations are specified by inference rules in the style of natural semantics. Preservation properties such as type and semantics preservation are discussed. The overall thesis of this paper is that suitable operator suites for automated adaptations and a corresponding transformational programming style can eventually be combined with other programming styles, such as polymorphic programming, modular programming, or the monadic style, in order to improve reusability of functional programs.

^{*}Partial support received from the Netherlands Organization for Scientific Research (NWO) under the *Generation of Program Transformation Systems* project

1 INTRODUCTION

Reuse is usually based on modularity, genericity and object-orientation. There are very powerful module systems for functional languages [9, 4]. State of the art functional languages are also very strong regarding language support for genericity, as provided, e.g., by polymorphism and polytypism. Furthermore, there are certain powerful ways of parameterization such as monadic programming [10, 14, 5] and arrows more recently.

This paper investigates the utility of program transformations to facilitate reuse of higher-order functional programs. The transformations illustrated in this paper are meant to "mimic" certain common adaptations performed by programmers manually. We want to provide evidence for the presumption that the need to anticipate the actual reuse from the start can be reduced by adding program transformation technology to the programming paradigm. The ultimate goal in this context would be to come up with a transformational style of programming based on a suitable operator suite for program adaptation. This paper just provides a few examples for automated adaptations. Our illustrative examples are concerned with adapting language interpreters, e.g., to introduce the monadic style in an interpreter. Inference rules in the style of natural semantics are used to formalize the transformations.

Most work on program transformations for functional programs focuses on implementation, particularly on compilation and optimization. Refer, e.g., to Pettorossi's and Proietti's survey on rules and strategies for transforming functional and logic programs [12] and issues of the annual ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation ([3],...). Transformations have been considered only to a very limited extent as a means of *extending* or *adapting* functional programs. The transformational approach suggested in this paper is partially based on our previous work on meta-programming for mainly *first-order* object languages [8].

The remaining paper is structured as follows. In Section 2, we will illustrate certain limitations for reuse in functional programming and we motivate some adaptations suitable to recover reusability. In Section 3, we develop a few transformation operators to automate the adaptations motivated before. The paper is concluded in Section 4. For brevity, related work is discussed throughout the paper rather than in a separate section.

2 MOTIVATION BY EXAMPLES

We perform a few experiments with extending language interpreters written in *Haskell*. We do not want to change the way how interpreters are developed. The domain has rather been chosen because it is a common domain for studying extensibility [14, 2, 5, 11, 1]. In programming practice, the illustrated limitations mean that programs cannot be reused without resorting to extra-linguistic features (i.e., text-editing) because the actual reuse has not been anticipated.

2.1 Preliminaries

We adopt the running example of [14]: the interpretation of certain constructs of a functional language. The following data types are needed in the rest of the paper:

type Name	=	String
data Exp	=	Zero Succ Exp Apply Exp Exp Var Name Lambda Name Exp
data Val	=	Wrong Nat Int Fun (Val \rightarrow Val)
type Env	=	[(Name, Val)]

Exp models abstract syntactical expressions. Val describes the values which can be encountered during interpretation including a special error value Wrong. Env models environments used in the interpretation of λs . An interpreter is a function that maps expressions in the context of an environment to a value.

2.2 Towards modularity

Let us attempt to provide a modular description of the basic interpreter. For that purpose Figure 1 presents two modules, *VAL* and *ENV*. The module *VAL* covers constructs which can be described at just the "value" level of semantic aspects as manifested by the profile $ie :: Exp \rightarrow Val$. The module *ENV* covers the remaining constructs which need to be interpreted in the context of an environment. Consequently, the profile $ie :: Exp \rightarrow Env \rightarrow Val$ is used.

module VAL	module ENV
$ie :: Exp \rightarrow Val$ $ie Zero = Nat 0$ $ie (Succ e) =$ $ie (Apply e_1 e_2) = apply (ie e_1) (ie e_2)$	<i>ie</i> :: Exp \rightarrow Env \rightarrow Val <i>ie</i> (Var <i>i</i>) $\rho = lookup \rho i$ <i>ie</i> (Lambda <i>i e</i>) $\rho =$ Fun λx . <i>ie</i> $e((i,x) : \rho)$
$apply :: \operatorname{Val} \to \operatorname{Val} \to \operatorname{Val}$ $apply (\operatorname{Fun} f) x = f x$ $apply f x = \operatorname{Wrong}$	$lookup :: Env \rightarrow Name \rightarrow Val$ $lookup [] i = Wrong$ $lookup ((j,v) : \rho) i = if i == j$ $then v$ $else lookup \rho i$

FIGURE 1. A very modular interpreter

Conceptually, this modularization looks perfect but soon it becomes clear that we cannot combine the two modules. The interpreter functions in both modules cannot be unified because of the different semantic aspects involved in their definition as reflected by the profiles. Consequently, a decomposition, where we try to abstract from environments in some part is not feasible. The module *VAL* cannot be reused in an interpreter with environments.

```
ie :: \operatorname{Exp} \to \boxed{\operatorname{Env}} \to \operatorname{Val}
ie \operatorname{Zero} \rho = \operatorname{Nat} 0
ie (\operatorname{Succ} e) \rho = \dots
ie (\operatorname{Apply} e_1 e_2) \rho = apply (ie e_1 \rho) (ie e_2 \rho)
```

FIGURE 2. Module VAL adapted to support environment propagation

We should comment on the possible employment of monads in this situation. If the modules *VAL* and *ENV* in Figure 1 *had been written* in the monadic style, and the environment *had been considered* as a part of the underlying monad in *ENV*, reuse would be possible. Further remarks regarding the monadic style of programming are given in the conclusion of the paper.

2.3 Making modules compatible

It is a rather simple adaptation to make the module *VAL* compatible with the module *ENV* by inserting parameters encoding environment propagation. The eventual result is illustrated in Figure 2. The corresponding transformation is developed in Subsection 3.1. After that adaptation, the modules can be combined provided the underlying module system allows recursive function and recursive type definitions to be broken up and spread across modules [4].

2.4 Migrating to the monadic style

The basic interpreter discussed so far could be extended to support error handling, states, I/O, profiling and others. In [14], such extensions are performed by means of monads. Therefore, a prerequisite is that programs are parameterized by a monad¹. Figure 3, for example, shows the monadic version of the module *VAL* from Figure 1. In Subsection 3.2 it is shown how to support the migration to the monadic style by a corresponding transformation. The transformation also allows one to extend the scope of a monad in a given program in a certain sense.

2.5 Adapting equations

Also if the monadic style is assumed, it is frequently necessary to perform adaptations which are beyond the scope of an ordinary functional language, that is to say, equations need to be modified to refine the computational behaviour.

Let us consider an example concerned with profiling. Suppose that function applications should be counted in the running interpreter example, i.e., applica-

¹A monad is represented in a language like *Haskell* as a type constructor *M* and two polymorphic functions $\uparrow :: \alpha \to M \alpha$ (usually called *unit* or *return*), $\gg :: M \alpha \to (\alpha \to M \beta) \to M \beta$ (usually called *bind*) obeying the common monad laws.

```
ie :: \operatorname{Exp} \to \underline{M} \operatorname{Val}
ie \operatorname{Zero} = \uparrow (\operatorname{Nat} 0)
ie (\operatorname{Succ} e) = \dots
ie (\operatorname{Apply} e_1 e_2) = ie e_1 \gg \lambda v_1. ie e_2 \gg \lambda v_2. apply v_1 v_2
apply :: \operatorname{Val} \to \operatorname{Val} \to \underline{M} \operatorname{Val}
apply (\operatorname{Fun} f) x = f x
apply f x = \uparrow \operatorname{Wrong}
```

FIGURE 3. Module VAL in monadic style (call-by-value)

tions of *apply*. The state monad is useful to hide the counter. Still one adaptation is necessary to increment the counter for applications of *apply*. The equation interpreting function applications has to be adapted as follows:

 $\textit{ie} (Apply e_1 e_2) = \textit{ie} e_1 \implies \lambda v_1. \textit{ie} e_2 \implies \lambda v_2. \boxed{\textit{tick} \implies \lambda().} \textit{apply } v_1 v_2$

A similar adaptation is described in the seminal paper on monads in functional programming [14], but presumably assuming that the programmer has to resort to extra-linguistic features for reuse. A rather rude method to solve the problems would be to replace the affected equation altogether (manually or by a transformation). A more disciplined notion of adaptation is discussed in Subsection 3.3. The notion is called *symbolic rewriting* because it allows us to replace a certain pattern of, for example, function applications in a functional program by another expression.

Another example of a rather subtle adaptation (also adopted from [14]) is concerned with error handling which is done so far using the special error value Wrong in Val. Let us suppose that proper error messages are required and that error handling should be modelled by the error monad with the type constructor $M \alpha = Ok \alpha$ | Fail *String*². It turns out that the monadic interpreter in Figure 3 cannot be used without further adaptation. The initial design decision to propagate errors based on a special error value Wrong in Val has to be altered. All the equations dealing with error situations are not appropriate any longer because—due to the migration to the monadic style—errors are accidentally represented as the "successful" value \uparrow Wrong = Ok Wrong. The last equation for *apply* has to be adapted, for example, as follows:

apply f x = Fail "ERROR: function expected."

3 TRANSFORMATION OPERATORS

The adaptations motivated above are automated as program transformations in this section. For brevity, the adaptations are discussed mostly at the level of terms and not at the level of complete functional programs. We assume familiarity with the

²i.e., an error message *s* is represented as Fail *s* and a "successful" value *v* is represented as $\uparrow v = Ok v$

simply-typed λ -calculus. In some cases, a more expressive calculus supporting polymorphism, data types and type constructors is needed, although that is not spelled out. The formal definition of the operators is shown in part using inference rules in the style of natural semantics.³

3.1 Propagation

A transformation operator \Rightarrow_{distr} , which can be used when a data structure in a functional program has to be propagated, is defined. In our running example, the operator can be used to make the module *VAL* compatible with module *ENV* as discussed in Subsection 2.3.

$\frac{x: \tau \in \Gamma x: \tau' \in \Delta x \Rightarrow_{pump}^{\Gamma; \tau; \tau'; t^+} t}{x \Rightarrow_{distr}^{\Gamma; \Delta; t^+} t}$	[D1]		
$\frac{x \notin \Delta}{x \Rightarrow_{distr}^{\Gamma;\Delta;t^+} x}$	[D2]	$t \Rightarrow_{pump}^{\Gamma;\tau;\tau;t^+} t t^+$	[P1]
$\frac{t \Rightarrow_{distr}^{\Gamma,x:\sigma;\Delta;t^+} t'}{\lambda x:\sigma. t \Rightarrow_{distr}^{\Gamma;\Delta;t^+} \lambda x:\sigma. t'}$	[D3]	$\frac{t x \Rightarrow_{pump}^{1,x;G;t;t';t'} t'}{t \Rightarrow_{pump}^{\Gamma;G \to \tau;t';t^+} \lambda x : \sigma. t'}$	[P2]
$\frac{t_1 \Rightarrow_{distr}^{\Gamma;\Delta;t^+} t_1' t_2 \Rightarrow_{distr}^{\Gamma;\Delta;t^+} t_2'}{t_1 \ t_2 \Rightarrow_{distr}^{\Gamma;\Delta;t^+} t_1' \ t_2'}$	[D4]		

FIGURE 4. Distribution of a term t^+

The operator \Rightarrow_{distr} is specified in Figure 4. The judgement $t \Rightarrow_{distr}^{\Gamma;\Delta;t^+} t'$ means the following. The term *t* is transformed into the term *t'* by the operator \Rightarrow_{distr} controlled by the parameters Γ , Δ and t^+ . The terms *t* and t^+ are assumed to be well-typed w.r.t. the type assignment Γ . t^+ is the term to be inserted by the transformation in order to encode propagation (e.g., the variable ρ in Figure 2). Finally, Δ is a special type assignment encoding the function symbols contributing to the propagation as follows. If *x* with $x : \sigma_1 \to \cdots \to \sigma_n \in \Gamma$ has to contribute to the propagation, then there exists an *i* such that $x : \sigma_i \to \cdots \to \sigma_n \in \Delta$ saying that applications of type $\sigma_i \to \cdots \to \sigma_n$ rooted by *x* should be extended by the further parameter t^+ . In this way, Δ also controls that t^+ should be inserted as the *i*-th parameter. The actual insertion of t^+ is initiated in rule [D1] and it is performed by the auxiliary operator \Rightarrow_{pump} defined in rules [P1] and [P2]. The transformation $x \Rightarrow_{pump}^{\Gamma;\tau,\tau',t^+} t_x$ "pumps up" $x : \tau$ to be become an application t_x of type τ' rooted by *x* with t^+ as the last parameter. The parameters before t^+ are

³We adopt some common conventions. The type judgement $\Gamma \vdash t : \tau$ is valid if the λ -term *t* has type τ w.r.t. the type assignment Γ . The extension of a type assignment Γ by $x : \tau$, as denoted by $\Gamma, x : \tau$, is valid if $x \notin \Gamma$. Terms are treated as equivalent up to renaming of bound variables (α -equivalence)

"handed over by wrapping" *x* with λ -abstractions. The remaining rules [D2] – [D4] descend into λ -terms.

Example 1. \Rightarrow_{distr} is illustrated for the right-hand side of the equation interpreting function applications in Figure 1, i.e., for the term *apply* (*ie* e_1) (*ie* e_2). To achieve the effect shown in Figure 2, the function *ie* should be extended by a parameter of type Env. We assume a free variable ρ serving as the parameter t^+ for that purpose. A suitable type assignment Γ is {*ie* : Exp \rightarrow Val, *apply* : Val \rightarrow Val \rightarrow Val, e_1 : Exp, e_2 : Exp, ρ : Env}. Finally, {*ie* : Val} serves as the parameter Δ . The resulting term is *apply* ((λx_1 : Exp. *ie* $x_1 \rho$) e_1) ((λx_2 : Exp. *ie* $x_2 \rho$) e_2). The resulting term has the same type as the original term w.r.t. a slightly different type assignment $\Gamma' = \{ie : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Val}, apply : \text{Val} \rightarrow \text{Val}, e_1 : \text{Exp}, e_2 : \text{Exp}, \rho : \text{Env}$ }.

It should be plausible that \Rightarrow_{distr} is semantics-preserving. One can show that the output program is not strict w.r.t. the inserted parameters. Moreover, it is possible to obtain the input program by just "projecting away" [8] the inserted parameters. \Rightarrow_{distr} is also type-preserving. This property indicates, for example, that an adapted term can be used in the context of the original term.

Let us consider a pragmatic problem with the transformation $t \Rightarrow_{distr}^{\Gamma;\Delta;t^+} t'$. The term t' will not be very readable because for each application of a symbol x, which is covered by Δ , $n - 1 \overline{\lambda}$ -abstractions are introduced, where n is the target parameter position of t^+ . This is a consequence of the simple way how parameters are inserted. To improve readability, more complex variants of \Rightarrow_{distr} and \Rightarrow_{pump} could be developed. Readability can also be recovered adopting partial evaluation for a subsequent simplification of t'. For that purpose, we assume that $\overline{\lambda}$ is used in λ -abstractions introduced by a transformation rather than λ .⁴ To make use of this convention, rule [P2] in Figure 4 has to be updated:

$$\frac{t x \Rightarrow_{pump}^{\Gamma,x;\sigma;\tau;\tau';t^+} t'}{t \Rightarrow_{pump}^{\Gamma;\sigma\to\tau;\tau';t^+} \overline{\lambda}x:\sigma.t'}$$
[P2]

Example 2. The term obtained in Example 1 can be simplified to *apply* (*ie* $e_1 \rho$) (*ie* $e_2 \rho$) by enforcing β -reduction for $\overline{\lambda}$ s.

3.2 Monad introduction

An approach to the introduction of a monad $\langle M, \uparrow, \rangle \gg \rangle$ in a given functional program is presented. Monad introduction is performed in two steps:

1. A sequential version of the given program is derived as an intermediate result by call-by-value sequencing; refer, e.g., to the *A*-normal forms in [7]. Conceptually, applications are flattened by means of (non-recursive) *let*-expressions. The notation *let* $x : \tau = t_1$ *in* t_2 can be represented in our framework without introducing additional syntax as the term ($\overline{\lambda}x : \tau$. t_2) t_1 .

⁴Such a convention is also used elsewhere, for example, in [7] to keep track of administrative λ -abstractions introduced by a CPS transformation.

2. A *let*-expression *let* $x : \tau = t_1$ *in* t_2 is transformed into $t_1 \gg \lambda x : \tau$. t_2 (i.e., \gg is used instead of *let*) if t_1 is supposed to be a computation. When values need to be coerced to computations, applications of \uparrow are inserted.

Functional programmers usually have to perform this kind of adaptation by hand. To use sequential programs as an intermediate representation is well in line with Moggi's computational meta-language [10].

Example 3. To illustrate sequencing, the right-hand side of the equation interpreting function applications in Figure 1 is flattened. Using *let*-notation, the corresponding sequential term is *let* $x_1 = e_1$ *in let* $x_2 = ie x_1$ *in let* $x_3 = e_2$ *in let* $x_4 = ie x_3$ *in apply* $x_2 x_4$.

We assume that sequencing is type- and semantics-preserving and partial evaluation can be used to nullify the effect of sequencing. The second step of monad introduction is modelled by the transformation operator \Rightarrow_{ms} specified in Figure 5. The operator \Rightarrow_{ms} and its collaborator $\Rightarrow_{ms'}$ are constrained by two type assignments, Γ for the input term and Γ' for the output term. The idea is that Γ' was derived from Γ by inserting some applications of *M*. Thereby, it can be controlled what functions should be computed in the monad. Another control parameter is the intended type of the output term. The terms are traversed by the auxiliary operator $\Rightarrow_{ms'}$, whereas \Rightarrow_{ms} possibly lifts the intermediate result returned by $\Rightarrow_{ms'}$ using \uparrow . Rules [M3] – [M6] simply descend into the input term. Rule [M7] transforms a let-expression $(\lambda x : \sigma, t_2) t_1$ into an application of \gg if t_1 gets a computation. One should not be confused that the input term pattern in rule [M6] is more general than the pattern of a let-expression covered by rule [M7]. The nondeterminism involved in Figure 5 is resolved by the type constraints. Note that partial evaluation can be used to remove the λ -abstractions remaining from the process of sequencing. We distinguish \Rightarrow_{ms} and $\Rightarrow_{ms'}$ to minimize the number of applications of \uparrow and \gg =, i.e., to avoid that the resulting program gets more sequential than necessary. Without this distinction, [M7] would tend to produce terms of the form $\uparrow t \gg \overline{\lambda}x : \sigma$. t' which can be simplified according to the monad law that \uparrow is the left unit of \gg =.

Example 4. The sequential term from Example 3 is transformed into the monadic style. For simplicity, only the function *ie* will be turned into monadic style in this example.⁵ The simplified monadic version of the sequential term is $(ie \ e_1) \gg \overline{\lambda}x_2$: Val. $(ie \ e_2) \gg \overline{\lambda}x_4$: Val. $\uparrow (apply \ x_2 \ x_4)$. The control parameters τ , Γ and Γ' for \Rightarrow_{ms} are as follows. $\tau = M$ Val because a right-hand side of an equation defining *ie* is considered. $\Gamma = \{ie : \text{Exp} \rightarrow \text{Val}, apply : \text{Val} \rightarrow \text{Val} \rightarrow \text{Val}, e_1 : \text{Exp}, e_2 : \text{Exp}\}$. Γ' is almost the same as Γ extended by the types for \uparrow and \gg . Only one application of *M* has to be inserted, namely $\Gamma' = \{ie : \text{Exp} \rightarrow (\alpha \rightarrow M \ \alpha) \gg (\alpha \rightarrow M \ \beta) \rightarrow M \ \beta\}$.

⁵That is sufficient, for example, if language constructs, whose interpretation involves states, are added. Note that in Figure 3 also the function *apply* is monadic. This would be necessary for other semantic aspects such as error handling.

$$\frac{t \Rightarrow_{ms'}^{\tau;\Gamma;\Gamma'} t'}{t \Rightarrow_{ms}^{ms'} t'} [M1] \qquad \frac{\Gamma' \vdash x:\tau}{x \Rightarrow_{ms'}^{\tau;\Gamma;\Gamma'} x} [M3]$$

$$\frac{t \Rightarrow_{ms}^{\tau;\Gamma;\Gamma'} t'}{t \Rightarrow_{ms}^{ms;\Gamma;\Gamma'} \uparrow t'} [M2] \qquad \frac{t \Rightarrow_{ms'}^{\tau;\Gamma;x;\sigma;\Gamma',x;\sigma} t'}{\lambda x:\sigma.t \Rightarrow_{ms'}^{\sigma\to\tau;\Gamma;\Gamma'} \lambda x:\sigma.t'} [M4]$$

$$\frac{t \Rightarrow_{ms}^{\tau;\Gamma;x;\sigma;\Gamma',x;\sigma} t'}{\overline{\lambda}x:\sigma.t \Rightarrow_{ms'}^{\sigma\to\tau;\Gamma;\Gamma'} \overline{\lambda}x:\sigma.t'} [M5]$$

$$\frac{t_1 \Rightarrow_{ms'}^{\sigma\to\tau;\Gamma;\Gamma'} t_1' t_2}{(\overline{\lambda}x:\sigma.t_2) t_1 \Rightarrow_{ms'}^{ms';\Gamma;\Gamma'} t_1' t_2} \Rightarrow_{ms'}^{ms;\Gamma;\Gamma',x;\sigma;\Gamma',x;\sigma} t_2' [M6]$$

FIGURE 5. Establishing a monad $\langle M, \uparrow, \rangle \gg = \rangle$

It can be verified that \Rightarrow_{ms} is semantics-preserving by unfolding the occurrences of M, \uparrow and \gg in Figure 5 according to the definition of the identity monad. It is interesting to notice that \Rightarrow_{ms} also can effectively be used to widen the monadic style by adapting further functions to be monadic. Thus, it does not matter, for example, that *apply* was not made monadic in Example 4 because it can be done later if necessary. However, the result of a repeated application of \Rightarrow_{ms} contains terms of the form $t \gg \overline{\lambda}x : \sigma$. $\uparrow x$ to be simplified to *t* according to the monad law that \uparrow is the right unit of \gg .

3.3 Symbolic rewriting

The general idea of symbolic rewriting is that terms in a given program are substituted to enhance or to adapt the computational behaviour. By such substitutions it is possible to perform several kinds of adaptations in the spirit of object-oriented features [13, 4] such as inheritance, mixins or meta-object protocols. The two problems illustrated in Subsection 2.5 can be handled by symbolic rewriting.

For brevity, we only consider a simple form of symbolic rewriting where applications *a* of a certain type τ and rooted by a certain symbol *x* are replaced by terms $(f \ a)$ of the same type.⁶ The corresponding judgements are of the form $t \Rightarrow_{wrap}^{\Gamma;x:\tau;f} t'$. Here, Γ is the type assignment for *t* and *f*. We might call this kind of adaptation wrapping and so *f* becomes the "wrapper" function.

Example 5. For the adaptation in Subsection 2.5 to count function applications, the following instance of \Rightarrow_{wrap} is needed: Γ is the type assignment for the monadic interpreter as in Example 4 extended by the type for the ticker function,

⁶Note that the adaptation of the definition of a function is more coarse grained because it corresponds to the adaptation of *all* applications of the function.

namely *tick* : *M* (). The applications to be adapted are described by *apply* : *M* Val. The wrapper function is $\lambda c : M \alpha$. *tick* $\gg \lambda$ (). *c*.

The formal specification of \Rightarrow_{wrap} is omitted for brevity. Although, the operator is certainly type-preserving (i.e., if $\Gamma \vdash t : \tau$, then $\Gamma \vdash t' : \tau$), it is in general not semantics-preserving. However, it is feasible to reason about semantics-preservation for certain patterns of replacements. In the example above we can use a notion of observable equivalence on programs by abstracting from the state.

4 CONCLUDING REMARKS

We discussed a few adaptations which can be modelled by program transformations. There are further adaptations which one can think of, e.g., adaptations based on the common folding/unfolding transformations originally proposed for the derivation of efficient programs [15]. The suggested transformational approach competes to some extent with other programming styles, particularly with modular programming and the monadic style. We rather think of it as a completion of the functional programming paradigm. Module systems are meant for programming in the large. However, they fail to support certain forms of refinements which are necessary to adapt a program before reuse or to alter design decisions. One of the obvious reasons for this restriction is that there should be good support for separate compilation. Programming for reuse in the monadic style requires that, in principle, all functions are monadic and that extensions are modelled as monad transformers. Extensive use of monads is suboptimal because it results in tangled, essentially sequential code, and all functions are concerned with the entire effect space. In [6], Filinski relaxes these requirements for at least Scheme-like languages. His approach is based on an extra (non-trivial) effect-typing system. In our approach, certain computational aspects can be modelled without resorting to monads. In the running example, it is indeed debatable if environment propagation is an effect to be modelled in a monad because there are functions which are not concerned with environments, e.g. apply. In our approach, there is also an incremental way to install a monad in a program and to extend the scope of it. Moreover, we are concerned with adaptations which go beyond the limits of monads as illustrated in the discussion on symbolic rewriting.

Our work also raises the following questions: What extensions of module systems might be useful? How could certain operators be integrated with existing module systems? Other topics for future work include the identification of further scenarios promoting our transformational approach, and the development of a *transformational* programming environment.

Acknowledgement I am very grateful for Guido Wachsmuth's experiments in *Coq* to verify some ideas presented in the paper. Thanks to Andrzej Filinski and Joe Wells for stimulating discussions. I thank Jan Heering, Jan Kort, Günter Riedewald and Joost Visser for comments on earlier versions of the paper.

REFERENCES

- E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 353–404. Springer, 1999.
- [2] R. Cartwright and M. Felleisen. Extensible denotational language specifications. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software: International Symposium*, volume 789 of *LNCS*, pages 244–272. Springer, Apr. 1994.
- [3] O. Danvy, editor. Proceedings of PEPM '99: The 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, Jan. 1999. BRICS Notes Series NS-99-1.
- [4] D. Duggan and C. Sourelis. Mixin Modules. In Proceedings of ICFP '96: The 1996 ACM SIGPLAN International Conference on Functional Programming, pages 262– 273, Philadelphia, Pennsylvania, 24–26 May 1996.
- [5] D. A. Espinosa. Semantic Lego. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1995.
- [6] A. Filinski. Representing Layered Monads. In Conference Record of POPL '99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, pages 175–188, New York, N.Y., Jan. 1999. ACM.
- [7] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *Proceedings of PLDI '93: The ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, 23–25 June 1993. *SIGPLAN Notices* 28(6), June 1993.
- [8] R. Lämmel. Declarative aspect-oriented programming. In Danvy [3], pages 131–146. BRICS Notes Series NS-99-1.
- [9] X. Leroy. Applicative Functors and Fully Transparent Higher-Order Modules. In Conference Record of POPL '95: The 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 142–153, San Francisco, California, Jan. 1995.
- [10] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, July 1991.
- [11] P. D. Mosses. Theory and Practice of Action Semantics. In Proceedings of MFCS '96: The 21st International Symposium on Mathematical Foundations of Computer Science, volume 1113 of LNCS, pages 37–61, Cracow, Poland, Sept. 1996. Springer.
- [12] A. Pettorossi and M. Proietti. Rules and Strategies for Transforming Functional and Logic Programs. ACM Computing Surveys, 28(2):360–414, June 1996.
- [13] E. Poll. Subtyping and Inheritance for Inductive Types. In Proceedings of TYPES'97 Workshop on Subtyping, inheritance and modular development of proofs, Durham, UK, September 1997.
- [14] P. Wadler. The Essence of Functional Programming. In Conference Record of POPL '92: The 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 1–14, Albequerque, New Mexico, Jan. 1992.
- [15] H. Zhu. How powerful are folding/unfolding transformations? *Journal of Functional Programming*, 4(1):89–112, Jan. 1994.