Towards Fully Controlled Overloading Across Module Boundaries

Stephan Herhut[1] and Sven-Bodo Scholz[2]

[1] Dept of Computer Science, University of Kiel, Germany
e-mail: sah@informatik.uni-kiel.de***
[2] Dept of Computer Science, University of Hertfordshire, United Kingdom
e-mail: S.Scholz@herts.ac.uk

Abstract. This paper proposes a set of modularisation constructs as well as a new implementation technique for overloading functions across module boundaries. In contrast to existing approaches, it allows to fully preserve separation of namespaces and it supports overloading of recursive functions in the context of subtyping, which in fact requires support for mutual recursion across module boundaries.
Based on a very simple applied λ-calculus as core language, the modularisation constructs are defined and a transformation scheme into an applied λ-calculus is presented. Furthermore, an outline of an implementation in the context of the functional programming language SaC is given.

1 Introduction

Most modern programming languages support some form of function overloading. In object-oriented languages such as Java[GJSB00] or Eiffel[Mey90] overloading is realised by means of inheritance and function overriding. In mainstream functional languages such as Haskell[Jon03] or Clean[PvE01] type classes provide the grounds for overloading. Other languages such as SaC[Sch03b] support overloading via subtyping. Combined with some form of modularisation support, overloading allows for extensive software reuse on one side and function customisation on the other.

More recently, techniques for generic programming have been introduced [Hin00,HP00,AP02,Sch03a]. These techniques lend themselves for creating compiler generated function instances. As a consequence, the number of function instances that is available increases, and for most argument types more than one instance exists.

Unfortunately, the existing module systems do not give the programmer full control over the applicability of individual function instances within individual modules. In particular, it is not possible to restrict the set of potential instances used in a given function application or to dynamically switch between two or more different overloadings that share instances. This may lead to undesired

*** most of the work was done during a research visit at[2].

program behaviour as imported functions that make use of overloaded functions may behave differently depending on the choice of instances in the actual context rather than the instances available in the module where they are defined.

In this paper, we propose a module system that gives the programmer exact control over the individual instances of overloaded functions. Each module provides local versions of overloaded functions that take into account only those instances that are locally visible. Depending on the way chosen for exporting such functions, these can or cannot be further extended by other modules. Since this extension process can be applied iteratively over several modules, in general, we obtain several versions of an overloaded function which share more than one instance. The proposed solution allows these versions to coexist at runtime; they may even share the code for individual instances which facilitates support for separate compilation.

The paper is structured as follows: the next section introduces a very simple functional language, essentially a first-order applied λ-calculus with overloading. It serves as an example language and helps in focusing on those aspects relevant to this paper, i.e., modules and overloading. Based on that language, section 3 introduces an example that serves as running example throughout the paper. It allows to demonstrate the locality problems involved in overloading across module boundaries. The next section informally introduces the proposed module system. It defines the syntactical extensions required and applies them to the running example. Section 5 formalises the semantics of the new constructs by specifying a transformation scheme for the module constructs into pure λ-calculus. Issues related to an implementation in the context of SaC are discussed in section 6. Related work is discussed in section 7 and some conclusions are drawn in section 8.

2 A simple applied λ-calculus with overloading

In this section, we introduce a simple functional language called $\mathcal{F}un$ which serves as minimised example language throughout the paper. As can be seen from the syntax description in Fig. 1, $\mathcal{F}un$ basically consists of an applied λ-calculus

$$
\begin{array}{lll}
Prg & \Rightarrow & \texttt{letrec\_ovld} \\
 & & \quad \left[\, Fun = Expr \,\right]^{*} \\
 & & \texttt{in}\ Expr \\
 & & \\
Expr & \Rightarrow & Var \mid Const \\
 & \mid & \lambda\, Var\, \texttt{.}\, Expr \\
 & \mid & (Expr\, Expr)
\end{array}
$$

Fig. 1. The syntax of $\mathcal{F}un$.

extended by built-in support for recursive definitions on the top-most level[1]. In contrast to the well known `letrec` construct we introduce a `letrec_ovld` construct. The only difference between these two constructs is the treatment of identical identifiers on the left hand side of definitions. In order to accommodate function overloading properly, such multiple definitions are considered separate instances of overloaded functions. Since we do not want to make any assumptions about how the function dispatch actually is implemented, we introduce an operation `ovld(` f_1`, ..., `f_n`)`. It defines a dispatch function for the (potentially overloaded) instances f_1 to f_n in an abstract way that neither precludes an implementation via an explicit dispatch function as found in [Kre03] nor an implementation via dictionaries as described in [WB89].

With this construct at hands, we can actually formalise the semantics of `letrec_ovld` constructs by defining a transformation into `letrec` constructs. Fig. 2 presents a transformation scheme $\mathcal{OVLD}$ to this effect. It consists of two

$$\mathcal{OVLD}\left[\!\left[\begin{array}{l}\texttt{letrec\_ovld}\\ \quad Fun_1 = Expr_1\\ \quad\vdots\\ \quad Fun_k = Expr_k\\ \texttt{in } Expr\end{array}\right]\!\right] = \begin{array}{l}\texttt{letrec}\\ \quad \mathcal{OVLD}\ [\{Fun_i = Expr_i | Fun_i \in \alpha_1\}\ ]\\ \quad\vdots\\ \quad \mathcal{OVLD}\ [\{Fun_i = Expr_i | Fun_i \in \alpha_m\}\ ]\\ \texttt{in } Expr\end{array}$$

where $\{\alpha_1, \ldots, \alpha_m\}$ is the set of equivalence classes of $\{Fun_1, \ldots, Fun_k\}$ with respect to equality of identifiers.

$$\mathcal{OVLD}\ [\![\ \{Fun_1 = Expr_1, \ldots, Fun_n = Expr_n\}\ ]\!]$$

$$= \begin{cases} Fun^{<1>}\ \texttt{=}\ Expr_1\\ \vdots\\ Fun^{<n>}\ \texttt{=}\ Expr_n\\ Fun\ \texttt{=ovld(}\ Fun^{<1>}\texttt{, ...,}\ Fun^{<n>}\texttt{)}\end{cases}$$

Fig. 2. The overloading scheme $\mathcal{OVLD}$.

steps. In a first step, all instances of an overloaded function are identified. This is shown in the upper part of fig. 2 where equivalence classes α_1 to α_m for identically named definitions are identified. Subsequently, each of these equivalence classes is transformed into a single overloaded function as shown in the lower part of fig. 2. Here, all instances of a function are renamed by introducing new unique identifiers for them, and a dispatch function is built using the `ovld` operator.

[1] Restricting built-in recursion to the top-level is not essential here. However, as we do want to introduce top-level modules only, this restriction simplifies all transformation schemes.

It combines the renamed functions into an overloaded function of the original name. After this transformation, the `letrec_ovld` construct can be replaced by well-known `letrec` construct.

It is important to notice here, that all function calls within the original program are now referring to the function containing the dispatch which actually ensures that all potential instances will be considered for each application of it.

3 The running example

Based on the language $\mathcal{F}un$ introduced in the previous section, this section presents an overloading example which contains most of the overloading-related problems in a nutshell. Fig. 3 presents a first part of it. On its left hand side, it

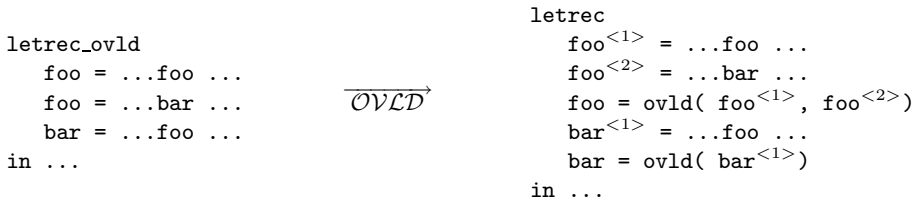

Fig. 3. Code within module A

contains the essentials of a $\mathcal{F}un$ program that provides two mutually recursive functions `foo` and `bar`. The function `foo` consists of two overloaded instances, whereas for `bar`, there exists only one instance. While the first instance directly calls itself, the second one does so indirectly by calling `bar`. The $\mathcal{OVLD}$ scheme presented in the previous section resolves the overloading as shown on the right hand side of fig. 3: The two instances of `foo` are renamed into `foo`$^{<1>}$ and `foo`$^{<2>}$ and a special overloading function `foo` is created. As the recursive calls on the right hand side remain untouched, they now refer to the dispatch functions. The function `bar` is processed in a similar manner.

Now, let us assume the code presented so far is located in some module A. Let us furthermore assume, that the function `foo` is to be extended by a further instance of it in some other module B. Fig. 4 presents the essential parts of such a module and their transformation according to the $\mathcal{OVLD}$ scheme. The `import`

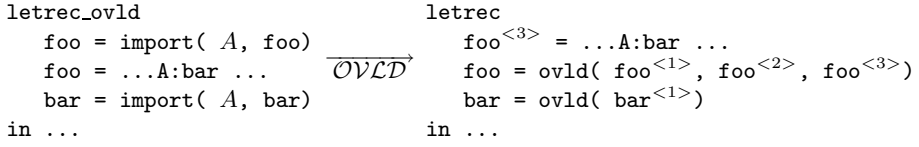

Fig. 4. Code within module B.

statements denote that the definitions of `foo` and `bar` from module A are to be imported into B.

The crucial design decision to be made here is to define which applied occurrences of `foo` should take the new instance into account. Given the recursive nature of `foo`, it would be desirable if at least the call in $\texttt{foo}^{<1>}$ would be aware of the new version. Since the call of `bar` in $\texttt{foo}^{<2>}$ eventually may lead to a recursive call of `foo` as well, the application of `foo` within `bar` should also be adapted. However, if `bar` is called directly, as indicated by the notion `A:bar`, the programmer may rely on the assumption that this function is not effected by further imports and their potential extension.

To accommodate this functionality, we basically need to allow for different, module specific "views" on overloaded functions. They are reflected by the dispatch functions contained in the individual modules. The call `A:bar` then leads to a change of view from B to A. Fig. 5 gives an overview of the potential call and dispatch graph within the two views. Dashed edges denote function applications,

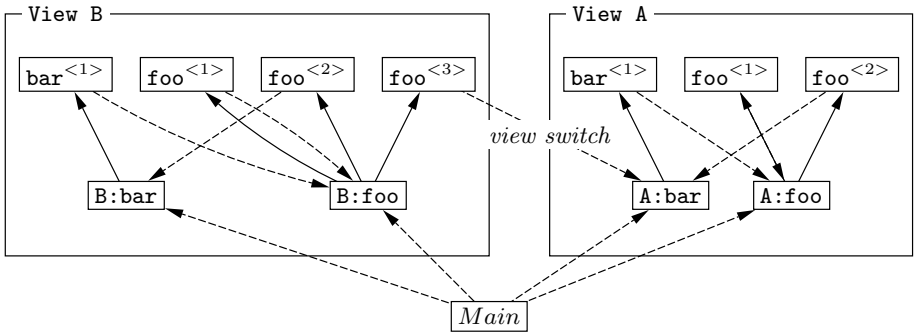

Fig. 5. Graph of possible function applications by `foo` from module B

while solid edges denote dispatch possibilities. On the right hand side, view A is presented. Within view A, only instances $\texttt{foo}^{<1>}$ and $\texttt{foo}^{<2>}$ are visible. Note here, that view A is completely self contained. Once an application of a function enters view A, all further function applications and dispatches stay within that view. In particular, there is no path from any function or instance within of view A to instance $\texttt{foo}^{<3>}$.

View B is given on the left hand side. It contains the additional instance $\texttt{foo}^{<3>}$. Therefore an application of `B:foo` can be dispatched to three different instances. In contrast to view A, view B is not self contained. The application of function `A:foo` within the third instance of `foo` leads to a switch of the current view towards view A. Note here, that although in B a new instance was added to the overloaded function `foo`, the local view of module A is still available. While the instances defined within A are visible within B, the instances added in B do not influence those being visible in A.

4 The module system

This section formally describes the syntax of the $\mathcal{F}un$ module system and applys it to the example given in the previous section.

$$
\begin{array}{lll}
Prg & \Rightarrow & \texttt{letrec\_ovld} \\
 & & \quad \left[\!\left[\; Import \; \right]\!\right]^{*} \\
 & & \quad \left[\!\left[\; Fun = Expr \; \right]\!\right]^{*} \\
 & & \texttt{in } Expr \\
 & & \\
Import & \Rightarrow & Var \texttt{ = use(} ModName\texttt{, } Fun \texttt{)} \\
 & | & \texttt{\{ } VarList \texttt{ \} = import(} ModName\texttt{, \{ } FunList \texttt{ \})} \\
 & & \\
Module & \Rightarrow & \texttt{letrec\_ovld} \\
 & & \quad \left[\!\left[\; Import \; \right]\!\right]^{*} \\
 & & \quad \left[\!\left[\; Fun = Expr \; \right]\!\right]^{*} \\
 & & \texttt{in export(\{ } FunList \texttt{ \}) provide(\{ } FunList \texttt{ \})}
\end{array}
$$

Fig. 6. Extended syntax of $\mathcal{F}un$

The module syntax in $\mathcal{F}un$ requires only a small extension to the basic syntax of $\mathcal{F}un$ as shown in fig. 6. The main difference between a program (Prg in fig. 6) and a module ($Module$ in fig. 6) is the lack of a goal expression, which is replaced by an interface declaration. By default, the scope of all functions defined in a module is local, i.e. limited to the module. To provide access from outside the defining module, a function has to be listed in the interface declaration within the set of provided or exported functions. Listing a function as provided admits access to it from outside the module but prevents from any further overloadings or the addition of its instances to another module. To admit the latter two options as well, the function has to be listed within the set of exported functions. We will refer to the former as providing a function and to the latter as exporting it.

Dual to the sets of provided and exported functions we introduce two operators, `use(` Mod`,` Fun `)` and `import(` Mod`, {` $FunList$ `} )`. The former enables access to a function Fun from a module Mod, provided Fun is enlisted in the module's interface. Note here, that it does not matter whether Fun is specified in the provided set or in the exported set. An import operation of the form `{` $VarList$ `} = import(` Mod`, {` $FunList$ `} )` has the same effect as if all instances of the functions in $FunList$ are defined locally as functions listed in $VarList$. All applications of functions in $FunList$ within these instances are replaced by their local counterpart. It is important to note here, that the sequence of imports `{` f_1 `} = import(` Mod`, {` f_1 `})` `{` f_2 `} = import(` Mod`, {` f_2 `})` is different from the single import `{` f_1`,` f_2 `} = import(` Mod`, {` f_1`,` f_2 `})`. The separate import of f_1 and f_2 allows to overload both, but possible ap-

plications of f_2 in f_1 and vice versa are not affected by new instances of that overloading. Importing both using a single import expresses the dependency between them so that applications of f_2 in f_1 and vice versa will be replaced by the corresponding local function.

Given the extended syntax, we can now specify our running example from section 2. Fig. 7 contains the listing of all three modules. Module A exports

```
A =  letrec_ovld
        foo = ...foo ...
        foo = ...bar ...
        bar = ...foo ...
     in export { foo, bar } provide {}

B =  letrec_ovld
        { foo, bar } = import( A, { foo, bar })
        A:bar = use( A, bar )
        foo = ...A:bar ...
     in export {} provide { foo, bar }

Main =  letrec_ovld
           foo = use( B, foo)
           bar = use( B, bar)
        in ...foo ...bar ...
```

Fig. 7. The module example using extended $\mathcal{F}un$.

it's instances of `foo` and `bar`, thereby granting access for further overloadings. Module B imports these again and defines a third instance of function `foo`. Furthermore, function `bar` is used as `A:bar`, Therefore the application of `A:bar` in the third instance of `foo` accesses only the first two instances of `foo` on the recursive application of `foo` in `A:bar`. Finally, `foo` and `bar` are provided by B. This limits the access from program $Main$ to using the provided functions and inhibits any further overloadings.

Given the above described behaviour, this program specification leads to exactly the graph of possible traces as given in fig. 5.

5 An operational semantics

This section formalises the semantics of the module constructs as introduced informally in the previous section. To do so, we specify a transformation scheme from the extended $\mathcal{F}un$ language into the applied λ-calculus without module constructs.

The basic idea is to parameterise modules by the overloaded functions that are exported. This allows for further instances to be brought in at a later stage from the outside. Technically, this requires all applications of exported functions

within the right hand sides of function definitions to be renamed by identifiers that are bound to freshly introduced λ-abstractions on the top most level. All that remains to be done apart from the renaming is to gather the provided and the exported functions in a way that is accessible from the outside in order to provide the interface of the module. This is achieved by placing them into customised records.

The complete $\mathcal{EXP}$ scheme is given in figure 8. For all functions Fun_i en-

$$
\mathcal{EXP}\left[\!\left[\begin{array}{l}
\texttt{letrec} \\
\quad Var_1 \texttt{ = } Import_1 \\
\quad \vdots \\
\quad Var_m \texttt{ = } Import_m \\
\quad Fun_1 \texttt{ = } Expr_1 \\
\quad \vdots \\
\quad Fun_n \texttt{ = } Expr_n \\
\texttt{in export(} \{\ Fun_1\ \ldots Fun_k\ \}\texttt{)} \\
\quad \texttt{provide(} \{\ Fun'_1\ \ldots Fun'_l\ \}\texttt{)}
\end{array}\right]\!\right]
$$

$$
\begin{array}{l}
= \lambda\,\overline{Fun_1}\ldots.\,\lambda\,\overline{Fun_k}\texttt{. letrec} \\
\qquad\qquad Var_1 \texttt{ = } Import_1 \\
\qquad\qquad \vdots \\
\qquad\qquad Var_m \texttt{ = } Import_m \\
\qquad\qquad Fun_1 \texttt{ = } [Fun_i \Leftarrow \overline{Fun_i}]Expr_1 \\
\qquad\qquad \vdots \\
\qquad\qquad Fun_n \texttt{ = } [Fun_i \Leftarrow \overline{Fun_i}]Expr_n \\
\qquad\quad \texttt{in } \{\ \ 'Fun_i' \texttt{ = } Fun_i\ \ \ldots\ \ 'Fun'_i' \texttt{ = } Fun'_i\ \ \}
\end{array}
$$

Fig. 8. The export resolution scheme $\mathcal{EXP}$.

listed in the set of exported functions, a new identifier $\overline{Fun_i}$ is introduced and bound to a λ-abstraction. To allow for substituting the actual instances of an overloading within applications on the right hand side of function definitions, these have to be renamed to the corresponding freshly introduced identifier. $[Fun_i \Leftarrow \overline{Fun_i}]Term$ denotes this substitution. Again, Fun_i represents all exported functions. As imports are handled in a different way, the substitution is only performed for functions that have been defined within the current module. Furthermore, the interface of the module is constructed. It consists of a record that contains all exported and provided functions. Note here, that the instances of an overloading are not directly exported as elements of the record. Instead, only the dispatch functions generated by applying the $\mathcal{OVLD}$ scheme are enlisted. As all existing instances are always contained within these functions, they are nevertheless accessible from outside the module.

On import, arguments have to be supplied for the parametrised modules created by the $\mathcal{EXP}$ scheme. Depending on the kind of import, the imported functions need to be customised to different views.

In case of a `use` operation, the imported functions need to be customised to the local view of the module where they are imported from. As the interface record of every module contains all these, this can be done by means of a self-application.

For `import` operations, the views have to be customised to that of the current module requiring an application to the local dispatch functions.

Fig. 9 gives a formal description of the transformation of imports by means of a scheme called $\mathcal{IMP}$. It is divided in two parts; the upper part defines a

$$\mathcal{IMP}\,[\![\, Var \texttt{ = use(} Mod \texttt{ , } Fun \texttt{)} \,]\!]$$

$$= \begin{cases} \overline{Var} \texttt{ = (} \lambda\overline{Exp_1}\texttt{.}\cdots\lambda\overline{Exp_m}\texttt{.}\, Body\ \overline{Var}.\texttt{`}Exp_1\texttt{`}\ \cdots\ \overline{Var}.\texttt{`}Exp_m\texttt{`}\ \texttt{)} \\ Var \texttt{ = } \overline{Var}.Fun \end{cases}$$

where $\lambda\overline{Exp_1}\texttt{.}\cdots\lambda\overline{Exp_m}\texttt{.}\, Body$ constitutes the definition of module Mod.

$$\mathcal{IMP}\,[\![\,\{\ Var_1\texttt{, }\ldots\texttt{, }Var_k\ \}\texttt{ = import(}Mod\texttt{ , }\{\ Fun_1\texttt{, }\ldots\texttt{, }Fun_k\ \}\texttt{)}\,]\!]$$

$$= \begin{cases} Var \texttt{ = (} \lambda\overline{Exp_1}\texttt{.}\cdots\lambda\overline{Exp_m}\texttt{.}\, Body\ filter(Exp_1)\ \cdots\ filter(Exp_m)\ \texttt{)} \\ Var_1 \texttt{ = } Var.\texttt{`}Fun_1\texttt{`} \\ \vdots \\ Var_k \texttt{ = } Var.\texttt{`}Fun_k\texttt{`} \end{cases}$$

where $\lambda\overline{Exp_1}\texttt{.}\cdots\lambda\overline{Exp_m}\texttt{.}\, Body$ constitutes the definition of module Mod and $Fun_{1\ldots k} \subseteq Exp_{1\ldots m}$ holds. $filter$ is defined as

$$filter(Exp) = \begin{cases} \overline{Exp}, & if Exp \in EFuns; \\ Exp, & if Exp \in Funs \setminus EFuns; \\ Var.Exp, & otherwise. \end{cases}$$

where $EFuns$ denotes the set of functions exported by the current module and $Funs = \{Fun_1, \ldots, Fun_k\}$ the set of imported functions.

Fig. 9. The import resolution scheme $\mathcal{IMP}$.

transformation for `use` operations while the lower part is dedicated to `import` operations. An import definition Var `= use(` Mod`,` Fun`)` is transformed into a pair of definitions. The first one creates a local view named $\overline{Var}$ of the module Mod. The view is created by applying the content of module Mod, i.e, an ex-

pression of the form $\lambda\,\overline{Exp_1}\,.\cdots\,\lambda\,\overline{Exp_m}\,.\,Body$ to the corresponding entries of the interface record of the local view $\overline{Var}$. To allow for this recursive application, record selection, denoted by $Var.`Exp`$, has to be lazy. In a second step, the imported function is selected from the resulting interface record and introduced as Var. Note here, that the view of module Mod is independent from the imported function Fun and the module which it is imported from. This in fact allows to reuse the view of a module, once defined, within several `use` statements across multiple modules. However, we do not do so to allow for simpler transformation rules.

The second part presents the scheme for transforming definitions of the form $\{\ Var_1\texttt{, ...,}\ Var_k\ \}$ `=` `import(` Mod`,` $\{\ Fun_1\texttt{, ...,}\ Fun_k\ \}$`)`. Again, Mod is replaced by the literal definition of the module Mod and an appropriate view for the imported module is constructed. The $filter$ function decides which view of a function has to be chosen as argument to the imported module. If the functions is imported and exported by the current module, $\overline{Exp}$ is chosen as parameter. Eventually, $\overline{Exp}$ is bound to a λ-abstraction on the top most level by the $\mathcal{EXP}$ scheme. If the function is imported but not exported again, the local view Exp of the current module is chosen. Finally, if the function is not imported at all, the local view $Var.`Exp`$ of the imported module is used.

In a second step, the specific functions are selected from the created view using the already described record selection. Note here, that all functions imported by one `import` statement share the created view. Thereby it is made sure that all applications of imported functions within the imported functions are replaced by the local versions. In contrast to views created by `use` operations, this view if specific to the import operation it was created for and connot be reused.

Given these schemes, the extended $\mathcal{F}un$ language can be transformed into the simple applied λ-calculus by applying $\mathcal{IMP} \circ \mathcal{OVLD} \circ \mathcal{EXP}$ to each module and program in the order of their definition.

6 Implementation

This section elaborates on how the module system introduced for $\mathcal{F}un$ in the proceeding section can be applied to an existing programming language. The programming language chosen for this purpose is SaC (Singe Assignment C) [Sch03b]. SaC is a functional language resembling C syntax extended by n-dimensional arrays as first-class language citizens and built-in support for overloading on functions based on subtyping.

The abstract `ovld` dispatch function in $\mathcal{F}un$ is resembled in SaC by special dispatch functions, referred to as wrapper functions. The wrapper functions decide which instance to use based on the (possibly non-static) type information known at runtime. Whenever possible, these wrapper functions are replaced by a static dispatch, or at least partially evaluated. A partially evaluated wrapper function is referred to as a specialisation. Details on SaC's semi-static dispatch scheme can be found in [Kre03].

In $\mathcal{F}un$ abstractions of exported functions on the top most level of modules are used to encode the current view. To implement this style of exporting functions, special function tables are used which hold the current wrapper function for each exported function. The entries have to be set to the wrapper function of the current view. Each application of an overloaded function is then indirected through this function table. This mimics the renaming operation used by the $\mathcal{OVLD}$ scheme for $\mathcal{F}un$. These two steps allow for dynamically customising the view of an overloading without the need of touching the once generated code of a module.

Within the λ-calculus representation of $\mathcal{F}un$ given above, this customisation is performed by selecting the function that has to be applied from a specially customised view of a module. Thus, whenever passing one of these special function applications, the current view on overloadings has to be updated. In SaC, instead of selecting the wrapper function from a previously generated view, the function tables are updated. The choice of entries resembles the selection rule for the arguments within the $\mathcal{IMP}$ scheme presented in fig. 9. Whenever a function is called that was introduced by a `use` operation, the function table is fully customised to the view of the module the function was imported from, i.e. the local wrapper functions.

Functions imported by an `import` statement lead to a partial update of the function table, only updating those entries, which have not been imported. All others are set to the view of the importing function. This includes leaving entries for exported functions untouched, as these already have been specialised by the call leading into the current module. This resembles the $filter$ function in the lower part of fig. 9.

This implementation technique allows for true separate compilation, as at compile time of a module only the signature of imported and used modules has to be known. There are no dependencies on code of other modules.

Unfortunately, by switching views through function tables, runtime overhead for the indirection and table updates is introduced. Furthermore, partial evaluation of wrapper functions as achieved by the semi-static dispatch is rendered impossible. In the setting of a compiler that relies on optimising runtime performance like SaC, this may be acceptable for rapid prototyping applications but it is not suitable for production runs of large applications.

To alleviate runtime overhead, specialisation of wrapper functions for specific views can be used. Whenever the local view of an overloaded function is statically known, a specialised wrapper function can be built and used instead of the generic version. This offers runtime improvement by removing indirections using the function table. Unfortunately, the view of a function call is only known in the case of used functions or on the top most level of imports, i.e. the main program. This limits the scope of such optimisation to few function applications.

To fully eliminate runtime overhead introduced by the presented overloading scheme, all views have to be made explicit. This approach closely mimics the $\mathcal{IMP}$ scheme presented for $\mathcal{F}un$. For all imported statements, a specialised version of the imported module has to be generated. As the view of a module can

be shared for all `use` statements, the amount of created views merely depends on the number of `import` statements. By generating the views explicitly, function tables and indirection of function applications are rendered unnecessary. Furthermore, the semi-static dispatch becomes applicable again as the wrapper functions are made explicit.

Of course, this rules out the ability of separate compilation, as the imported module has to be recompiled on every import and a strong dependency on the code of the imported module is introduced. This heavily increases the time needed for recompilation of applications whenever the code of a module is changed. In the setting of long running applications the improved runtime performance outbalances this downside.

7 Related work

The work presented here directly relates to the module systems of Haskell and Clean both of which support function overloading across module boundaries.

Haskell combines overloading based on type classes[HHPW96] with a module system supporting namespace separation and information hiding[DJH02][2]. In principle, Haskell supports only extendable imports similar to the `import` and `export` operations described in this paper. Different views on overloaded functions or non-extendable versions as they can be generated by the `use` and `provide` operations are not supported. However, in Haskell, the import is not restricted to those symbols that are explicitly specified. Instead, the transitive closure of the call graph is imported including all potentially used instances of overloaded functions.

The Clean programming language has no support for namespaces, but generates module locality by means of scopes. Every module has a dedicated interface description, allowing to hide functions outside of the module's implementation. Imports in Clean behave in the same way as the `import` operation described here, i.e., instead of including the transitive closure of the call graph each function (instance) has to be imported explicitly. Despite this similarity with the approach presented here, Clean does not support any form of non-extendable overloading or shared instances between different overloadings.

A related problem to overloading across module boundaries is spanning recursive functions across several modules. So called mixin modules[DS96] have been proposed as a means to provide this functionality in Standard ML[MTH90]. Instead of a dispatch between several instances of a function, pattern matching on the arguments is used to select a target function. To allow for extending the set of possible target functions, a special `inner` pattern is added. Extensions to the pattern matching across several mixins are then merged along the special `inner` pattern. In contrast to our approach, a specific view on the recursive func-

[2] We are not aware of a specification of Haskell's module system wrt. overloading across module boundaries. The account given here is based on two actual implementations of Haskell, GHC and HUGS.

tion has to be specified by joining mixins to a STANDARD ML module. Once the mixins have been transformed into a module, no further extension is possible.

8 Conclusions

Overloading across module boundaries usually conflicts with the principle of locality. If used intensively, spurious overloadings and unexpected program behaviour may be the consequence.

In this paper, a module system is proposed that gives the programmer explicit control over the exact set of instances of overloaded functions that are visible within individual modules. This is achieved by means of two different export/import mechanisms: one that enables external usage only, and another one that allows later extension via further overloading. Combined, these mechanisms can be utilised for creating several different "views" on overloaded functions each of which can be fine-tuned to the needs of individual applications.

Although this requires more elaborate dispatch techniques, it is shown that separate compilation still is feasible in the proposed approach. As this genericity comes for some extra cost at runtime, the paper outlines alternative implementations based on code specialisation ond partial evaluation. They give up separate compilation to some extent, but allow for improved runtime performance. In the context of SAC, this may entirely eliminate overhead due to increased dispatch complexity.

References

[AP02] A. Alimarine and R. Plasmeijer. A Generic Programming Extension for Clean. In T. Arts and M. Mohnen, editors, *Proceedings of the 13th International Workshop on Implementation of Functional Languages (IFL'01), Stockholm, Sweden, selected papers*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–186. Springer-Verlag, Berlin, Germany, 2002.

[DJH02] Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the haskell 98 module system. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 17–28. ACM Press, 2002.

[DS96] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 262–273. ACM Press, 1996.

[GJSB00] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java Language Specification*. Java series. Addison-Wesley, Reading, Massachusetts, second edition, 2000.

[HHPW96] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[Hin00] R. Hinze. *Generic Programs and Proofs*. Habilitation thesis, Universität Bonn, 2000.

[HP00] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Proceedings of the 4th Haskell Workshop*, 2000.

[Jon03] S.L. Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, Cambridge, UK, 2003.

[Kre03] D.J. Kreye. *A Compiler Backend for Generic Programming with Arrays.* PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 2003.

[Mey90] Bertrand Meyer. *Eiffel: The Language.* Prentice Hall, 1990.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990. ISBN 0-262-63132-6.

[PvE01] R. Plasmeijer and M. van Eekelen. *Concurrent Clean 1.3.1 Language Report.* High Level Software Tools B.V. and University of Nijmegen, 2001.

[Sch03a] Sven-Bodo Scholz. *Generic Array Programming.* Habilitation thesis, Universität Kiel, 2003.

[Sch03b] Sven-Bodo Scholz. Single Assignment C — efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[WB89] P. Wadler and S. Blott. How to Make ad-hoc Polymorphism Less ad hoc. In *POPL '89*, pages 60–76. ACM Press, 1989.