Reflections in Opal –

Meta Information in a Functional Programming Language

Klaus Didrich, Wolfgang Grieskamp, Florian Schintke, Till Tantau and Baltasar Trancón-y-Widemann

Technische Universität Berlin {kd,wg,schintke,tantau,bt}@cs.tu-berlin.de

Abstract We report on an extension of the Opal system that allows the use of *reflections*. Using reflections, a programmer can query information like the type of an object at runtime. The type can in turn be queried for properties like the constructor and deconstructor functions, and the resulting reflected functions can be evaluated. These facilities can be used for generic meta-programming. We describe the reflection interface of Opal and its applications, and sketch the implementation. For an existing language implementation like Opal's the extension by a reflection facility is challenging: in a statically typed language the management of *runtime type information* seems to be an alien objective. However, it turns out that runtime type information can be incorporated in an elegant way by a source-level transformation and an appropriate set of library modules. We show how this transformation can be done without changing the Opal core system and causing runtime overhead only where reflections are actually used.

Contents

Introduction

Reflection Interface . . .

Semantic Foundation

Implementation

Conclusion

References

Contents

1 Introduction 3

2 Reflection Interface and Applications 5

2.1 Core Interface 5
2.2 Inspection Interface 11
2.3 Lookup Interface 14

3 Semantic Foundation 18

3.1 Algebraic Foundation 18
3.2 Syntactic Transformation 19

4 Implementation 20

4.1 Construction of Value Reflections 20
4.2 Free Type and Function Reflection 21

5 Conclusion 23

Contents

Introduction

Reflection Interface . . .

Semantic Foundation

Implementation

Conclusion

References

1. Introduction

Modern functional languages support a powerful, efficient, and safe programming paradigm based on parametric polymorphism and higher-orderness in conjunction with static type discipline. However, the advantages of static typing – safety and efficient execution – are paid for by less flexibility regarding *generic meta programming*. High-level environments for languages that allow meta programming like LISP or Smalltalk are therefore traditionally based exclusively on dynamic typing.

The integration of *dynamic types* into a static, parametric-polymorphic type discipline has been investigated in [3,9,2] and is nowadays well understood. However, dynamic types are only one prerequisite for generic meta programming. To utilise dynamically typed objects, it is necessary to *reflect* information about objects and types. Information of interest includes *full instantiation information* for reflected values in polymorphic contexts, and the *free type definition* of given types. The reflection of a free type definition consists of reflected knowledge about its constructor, discriminator and deconstructor functions. This knowledge can be used to process the free type's elements generically.

As rule of thumb, reflections make available as much compile time information as possible to the program. Code based on this additional information may be executed at runtime or, using partial evaluation, at compile time. Reflections in this sense became well known as part of the core technology of Java [12].

In this paper we report on a pragmatic extension of the Opal language and compilation system that allows the use of reflections. The extension is in fact quite moderate. We add one further keyword which is defined by a purely syntactic transformation into regular Opal. A set of library structures implements the reflection mechanism by connecting the running Opal program to compiler generated resources. The library is in turn based on a hand-coded module of the Opal runtime system, that allows dynamic linking and execution of reflected functions.

This paper is organised as follows. We first discuss the design from the *application view* of reflections as provided by our extension. We then discuss the *semantic foundation* of reflections as used in Opal. The semantic foundation is given by a syn-

tactic transformation, and therefore in fact also prepares the *implementation* which is discussed last.

Background. Opal [4,10,1] is a strongly typed, higher-order, strict and pure functional language. It can be classified alongside ML, Haskell, and other modern functional programming languages. However, the language also has a distinctively algebraic flavour in the tradition of languages such as CIP-L, OBJ and others. Instead of ML-style polymorphism the language provides parameterised modules, called *structures*, which are comparable to Ada's generics or to C++ templates. As in C++ instantiation of parameterised structures is automatically inferred as far as possible from context information.[1] Opal has a comprehensive implementation, including a compiler which produces very fast code [8], and a large library of reusable software components.

We will use Opal in this paper on a level that should be intelligible to readers familiar with standard concepts of functional languages but who have no specific knowledge of the Opal language. Explanations of unusual Opal constructs will be given as we proceed. In the conclusion we will justify that the introduced Opal concepts can also be expressed in functional languages with type classes such as Haskell.

[1] Since structures may not only be parameterised by types but also by constant values, complete inference of instantiations is not possible.

2. Reflection Interface and Applications

This section describes how reflections are used in Opal. The *core interface* provides dynamic types with full instantiation information. The *inspection interface* is used to create reflections of free type definitions. Finally, the *lookup interface* is used to convert symbolic information into reflections. As a running example, a generic printer is constructed incrementally using each of the interfaces. The most developed form of the generic printer will be able to convert an arbitrary value into a human readable string *without any special preparation by the programmer*. Specifically, even for recursive user-defined types no conversion functions need to be provided as is necessary in most languages including C++.

2.1. Core Interface

The basic interface to the reflection mechanism is provided by two Opal structures (modules). `Reflection` declares the types and `ReflectionBuild` the functions which make up the core functionality.

Representing Reflections. As the reflection system should make available compile time information at runtime, it must define several types which *model* the compile time information. These types are used to talk *in* the programming language Opal *about* the programming language Opal.

In Figure 1, the data model of the core reflection system is presented as a collection of free type definitions.[2] The meaning of the definitions in Figure 1 are in detail:

[2] The keyword `SIGNATURE` marks the export interface of an Opal structure. For each variant of a free type, marked by the keyword `TYPE`, a constructor function, a discriminator function and deconstructor (partial selector) functions are automatically introduced. An example discriminator function is `FUN sortKind? : kind -> bool`, an example deconstructor is `FUN factors: type -> seq[type]`.

Figure 1 Core Interface: Types

```
SIGNATURE Reflection
  TYPE name   == name      (identifier : string,
                            kind       : kind,
                            struct     : struct)
  TYPE kind   == sortKind
                 valueKind (type       : type)
  TYPE sort   == sort      (name       : name)
  TYPE type   == basic     (sort       : sort)
                 product   (factors    : seq[type])
                 function  (domain     : type,
                            codomain   : type)
  TYPE struct == struct    (identifier : string,
                            instance   : seq[name])
  SORT value
  FUN  type : value -> type
```

Reflections in Opal

Contents

Introduction

Reflection Interface . . .

Semantic Foundation

Implementation

Conclusion

References

name: A name determines a named object in an Opal program, and is described by its *identifier*, *kind* and *origin structure*. Note that Opal employs full overloading, such that all of these components are required in order to identify a name uniquely. Opal employs the following syntax for names:

```
identifier'origin structure[instantiation parameters] : kind
```

The origin structure, the list of instantiation parameters and the kind are optional if they can be derived uniquely from context. The kind can be `SORT` or a functionality. Example Opal names are `int : SORT` and `+'Int : int ** int -> int`.

kind: A name represents either a sort or a value of a specific type. These two possibilities are distinguished by the name's `kind`. For example, the name `int'Int : SORT` in literal Opal syntax, is reflected by the name `name("int", sortKind, IntStruct)` meaning that `int` is a sort declared in some structure `Int`. Here, `IntStruct` is in turn a reflection of the structure `Int` as discussed below.

sort: A sort is uniquely identified by its `name`.

type: An Opal type, which can be either basic, a Cartesian product or a function space, is reflected by `type`. If `IntName` is the above name for the integer sort, then the name of the function `+` declared in the structure `Int` is in all glory detail:[3]

```
name("+", valueKind(function(product(basic(sort(IntName)) ::
                                     basic(sort(IntName)) :: <>),
                             basic(sort(IntName)))),
     IntStruct)
```

[3] `<>` denotes the empty sequence in Opal and `::` denotes the cons function which can be written in infix.

Figure 2 Core Interface: Functions

```
SIGNATURE ReflectionBuild[alpha]
  SORT alpha
  ASSUME Dynamic[alpha]
  FUN reflect     : alpha      -> value
  FUN reflects?   : value      -> bool
  FUN content     : value      -> alpha
```

`struct`: An Opal structure is determined by its *identifier* and an *instantiation list*. An instantiation list is a sequence of names. It is empty if the structure has no parameters. For example, the structure `Int` is represented by `struct("Int", <>)` which we labeled `IntStruct` earlier. The structure `Seq[int]` is represented by `struct("Seq", IntName :: <>)`

`value`: A `value` is a reflection of a value. It stores the type of the reflected value as well as the value itself. However, the value is stored in an opaque way and cannot be observed directly. To achieve opaqueness, `value` is not defined as free type using `TYPE`, but as sort using `SORT`.

Constructing Reflections. The central functionality of the core reflection system is the translation between runtime values and reflections. This is accomplished by the functions declared in the structure shown in Figure 2 which is parameterised over the sort `alpha`. A parameterised Opal structure is a section of declarations that are uniformly polymorphic over the parameters listed in the formal parameter list.

The `reflect` function takes an arbitrary value and converts it into a reflection. Thus if `5` has type `int`, then `type(reflect(5))` delivers an object of type `type` describing the type `int`. The function `reflects?` tests whether a value is of a certain type. The type is given by the instance of the generic parameter, such that we have for example `reflects?[int](reflect(5)) = true` and `reflects?[bool](reflect(5)) =`

`false`. Finally, `content` is a partial function that extracts the representation from a reflected value. We have `content[int](reflect(5)) = 5`, whereas `content[bool](reflect(5))` is undefined.

Note that in difference to dynamic types as described in [9,2], the type of a reflected value always contains the full instantiation information. Consider the definition of a `typeof` function:

```
IMPLEMENTATION TypeOf[alpha]
  ASSUME Dynamic[alpha]
  FUN typeOf : alpha -> type
  DEF typeOf(x) == type(reflect(x))
```

A call `typeof(5)` yields the reflected type `int`, and a call `typeof(true)` the reflected type of Booleans. Other approaches to the combination of parametric polymorphism and dynamic types insist on returning the (constant) *type scheme* `alpha`.

The `ASSUME Dynamic[alpha]` declaration in the signature of `ReflectionBuild` does all the magic which enables the above functionality. Before we take a closer look at it, we give an example that illustrates the usage of the core interface.

Example: Virtual Methods. An interesting application of functions which behave differently for different types are virtual methods in object-orientated programming. The code executed upon their invocation depends on the actual type of an object at runtime. In object-orientated programming languages, the mapping process from types to code during method invocation is performed automatically and is normally hidden from the programmer. Using reflections, a method invocation protocol can be implemented in Opal that mimics the built-in behaviour of virtual methods in object-orientated programming languages.

The following example shows how a generic printing function might be implemented using virtual methods. The function `::` is used to add a function to a method, thus adding a new behaviour to the method – its implementation will be described later on.

```
IMPLEMENTATION Print [alpha]
  ASSUME Dynamic[alpha]
  IMPORT Method ONLY ::     -- uninstantiated import (instances
                            -- will be automatically derived)
  FUN default   : alpha -> string
  DEF default (a) == "some value"
  FUN printBool : bool -> string
  FUN printInt  : int  -> string

  FUN print : alpha -> string
  DEF print == printBool :: printInt :: default
```

The constructed method `print` can be invoked by calling `print(true)` to print a Boolean value or by `print(5)` to print the number five. Note that the above implementation is type safe. It is guaranteed that a function like `printBool` is never applied to anything but a Boolean value.

The implementation of the method constructor `::` is surprisingly simple. It takes a function and a method and returns a new method, that calls the new function whenever its input has the type expected by the new function and calls the old method otherwise:

```
IMPLEMENTATION Method [alpha, special, result]
  ASSUME Dynamic[alpha] Dynamic[special]
  FUN :: : (special -> result) ** (alpha -> result) -> (alpha -> result)
  DEF (func :: method)(a) ==
    IF reflects?[special](r) THEN func(content[special](r))
                             ELSE method(a) FI
    WHERE r == reflect(a)
```

Above, the `a` of type `alpha` represents the parameter of the method. If the reflection `r` of `a` also reflects an object of type `special`, then we can safely convert `a` to type `special` and call `func`. Otherwise the old method is tried.

It is remarkable that in this example reflection types appear only on the level of library programming – here in the structure `Method`. In the structure `Print`, we only have to mention the assumption that its parameter has to be "dynamic".[4] This *abstraction* from the use of core reflection becomes possible because even for polymorphic parameters full type information is available.

What does the assumption `ASSUME Dynamic[alpha]` mean? Pragmatically, it just indicates that *runtime type information* (RTTI) somehow needs to be made available for instances of the parameter `alpha`. Every instantiation of a structure, where `Dynamic` assumptions are made on a parameter, must satisfy this assumption by providing RTTI for the parameter. If a structure is instantiated with a basic type, this is easy to achieve since for types such as `int` the compiler can literally construct the runtime type information. If a structure is instantiated with a formal parameter of another structure, then the assumption can be resolved if there is a similar assumption in the instantiating structure. This is, for example, the case for the formal parameter `alpha` in `Print` which is (implicitly) passed to `Method`.

The `ASSUME Dynamic[alpha]` concept is the one and only extension we need to add to the Opal language (but a generalisation of this construct is part of the forthcoming Opal-2 language [5]). The construct is similar to conditional polymorphism provided by Haskell's type classes [7], and `Dynamic` can be modeled in Haskell as a builtin type class whose instances are automatically generated by the compiler. This will be discussed in the conclusion.

2.2. Inspection Interface

The core reflection interface presented in the previous section provides *dynamic types*. But the reflection mechanism goes further. In addition to types it permits the reflection of *properties* of objects, the construction and deconstruction of tuples and the application of reflected functions to reflected values.

[4] In fact even this assumption is not necessary, since it can be derived from the import of the structure `Method` – it is just for documentation purposes.

Figure 3 Inspection Interface

```
SIGNATURE ReflectionInspect
  TYPE variant    == variant (constructorName : name,
                              constructor     : value -> value,
                              discriminator   : value -> bool,
                              deconstructor   : value -> value)
  FUN freeType?   : sort           -> bool
  FUN variants    : sort           -> seq[variant]
  FUN applicable? : value ** value -> bool
  FUN apply       : value ** value -> value
  FUN tuple       : seq[value]     -> value
  FUN untuple     : value          -> seq[value]
```

Inspecting Reflections. The structure `ReflectionInspect` shown in Figure 3 introduces types and functions for inspecting free type definitions. Given a sort `s`, the function `freeType?` tests whether its argument has an associated free type definition. The partial function `variants` delivers the variants for sorts that are free types. A variant is described by a quadruple of the variant's constructor function's name and a set of functions working on reflections. The `constructor` function in a variant of the sort `s` takes an appropriately typed argument and constructs a new value of type `s`. The Boolean `discriminator` in a variant function tests whether its argument is constructed from this variant. Finally, the `deconstructor` function in a variant decomposes a value into its components; it is undefined if the passed value is not constructed by this variant.

The `tuple` and `untuple` functions construct and deconstruct reflected tuple values, including the empty tuple. Using `applicable?(f,x)` one can test whether the reflected value `f` is a function that can be applied to the argument `x`. The expression `apply(f,x)` yields the result of this application and is undefined if `f` is not applicable to `x` due to typing conditions.

Note that due to syntactic restrictions of Opal's parameterisation, it is not possible to call functions directly in the style `content[A->B](r)(x)`. In Opal, structures may be only be instantiated by names, not by types.[5] Similarly, tuples cannot be reflected directly using `reflect[A ** B](x)`. However, even if this restriction were not present the tuple, untuple, and apply functions would still be essential, since they allow writing code generic over the number of parameters of a reflected function value.

Example: Generic Printing. As an example of the application of the inspection interface, the printing function introduced in the previous section is extended. A more powerful default printing method takes over if no type-specific function is found in the method. We redefine the function `default` from structure `Print` as follows:[6]

```
DEF default(x) ==
  LET
    r == reflect(x)
    t == type(r)
  IN IF basic?(t) ANDIF freeType?(sort(t))
     THEN print(constructorName(vari)) ++
                format(print * untuple(deconstructor(vari)(r)))
          WHERE vari == find(\\ v . discriminator(v)(r),
                             variants(sort(t)))
     ELSE "some value of type: " ++ print(t)
     FI
```

Above, we construct a reflected value `r` from `x` and test whether its type has a free type definition. If so, we search for a variant whose discriminator is true for the given value

[5] This Opal deficiency stems from its origin in algebraic specification theory and is removed in the forthcoming Opal-2.

[6] In Opal, `\\ v .` denotes the lambda operator $\lambda v.$, the star `*` denotes the mapping operator and `++` denotes string concatenation.

(there must exist exactly one). We then recursively apply `print` to the name of the variant and all elements of the deconstructed component sequence. The auxiliary function `format` takes a sequence of strings and writes them alongside each other, delimited by commas and enclosed by parentheses.

There remains a problem with the above implementation. The `print` method for deconstructed components of a value is called for *reflections*, and these reflections will be printed instead of their contents. Though it is in principle legal to reflect reflections, this is not the expected behaviour here. Rather, one would expect a method like `print` to work on the encapsulated value if a reflection is passed. The problem is fixed by modifying the method build function `::` as follows:

```
DEF (func :: method)(a) ==
  IF reflects?[special](r) THEN func(content[special](r))
                           ELSE method(a) FI
  WHERE r == IF reflects?[value](reflect(a))
             THEN content[value](reflect(a))
             ELSE reflect(a) FI
```

If the parameter `a` is a reflection of a `value` it is taken as is, otherwise a new reflection is constructed. The same modification has to be applied to the `default` method.

2.3. Lookup Interface

An important functionality of a fully-fledged reflection system is the ability to lookup reflected values from symbolic, textual representations. This allows the dynamic binding of code to a running program. For example, a device driver or a component plug-in might be loaded at runtime. A compiler could be integrated this way as well: Runtime generated code could be compiled and then bound to the running program in a safe way using symbolic lookup.

Figure 4 Lookup Interface

```
SIGNATURE ReflectionLookup
  IMPORT Com COMPLETELY   -- com'Com is Opal's IO Monad
  FUN extern    : string -> com[set[name]]
  FUN intern    : string -> set[name]
  FUN bind      : name   -> value
```

Lookup Functions. The structure `ReflectionLookup` shown in Figure 4 declares the basic commands for symbolic lookup. The command `extern` takes the symbolic representation of a (partial) name in Opal syntax and computes the set of all names that match the symbolic representation in a code repository. This command is monadic since it depends on side-effects. The code repository can be changed dynamically by the environment or by the program itself – for example, by downloading a code component from the Internet. The code repository will be discussed in more detail in Section 4. The function `intern` behaves similarly, but it can lookup only names that have been statically linked with the program. Therefore it can be a pure function.

For instance, `extern("int'Int")` searches for the name of the sort `int` in the structure `Int`. As it is possible that several names match a symbolic representation due to overloading, `extern` and `intern` return sets of names. For example, `extern("-'Int")` returns both the name of the unary minus as well as the name of the difference function defined in the structure `Int`. To narrow down a search, one can specify a functionality like in `extern("-'Int : int -> int")`. The rules for name resolution are similar to those in the Opal language. The resolution of types is performed in the context of the structure associated with the main identifier searched for: thus `int -> int` above is promoted to `int'Int -> int'Int`. The main identifier must always be supplied with a structure name, such that `extern("asString: foo'Foo -> string")` is not valid.

For parameterised structures the instantiation has to be specified. Thus `intern ("seq'Seq[int'Int]")` is valid, whereas `intern("seq'Seq")` is not. The given instantiation parameters must fit the formal ones of a structure according to the rules of the

Opal language. If there are any `Dynamic` assumptions on the parameters of a structure they are satisfied automatically, which is possible since only basic instances of parameterised structures are dealt with.

Once a name representing a value has been constructed, i.e. a name having `valueKind`, it can be *bound*. Applying the function `bind` yields the desired reflected value which can be further processed by the inspection functions.

Example: Customised Printing. We extend the `default` method for printing values by adding symbolic lookup of customised formatting functions. The lookup is based on naming conventions: every function originating from the structure of the sort of the printed value which is declared as `format : sort -> string` is considered as a candidate. We get the following code:[7]

```
DEF default(x) ==
  LET r == reflect(x) IN
  IF basic?(type(r)) THEN
     LET sym == "format'" ++ print(struct(name(sort(type(r)))))
                          ++ " : " ++ print(sort(type(r)))
                          ++ " -> string"
         cands == intern(sym)
     IN
     IF empty?(cands)
     THEN oldDefault(r)
     ELSE content[string](apply(bind(arb(cands)), r))
     FI
  ELSE oldDefault(r) FI
```

This example just illustrates the capabilities of symbolic lookup – we do not propose that printing should actually be defined this way, since it is methodological questionable to do such lookups based on pure naming conventions. Instead a further extension of

[7] `empty?` checks whether a set is empty and `arb` returns an arbitrary element if it is not.

the reflection mechanism, discussed in the conclusion, might be used which allows the reflection of assumption and theories.

Example: Plugins. As an example for the usage of the lookup command `extern`, consider the implementation of a dynamically loadable plugin with an interface described by the type `plugin`. Loading the plugin is then achieved as follows:[8]

```
TYPE plugin == plugin(init: com[void], oper: input -> com[output])
FUN loadPlugin : string -> com[plugin]
DEF loadPlugin(ident) ==
  extern("init'" ++ ident)  & (\\ inits .
  extern("call'" ++ ident)  & (\\ calls .
  LET
    init == bind(arb(inits))
    call == bind(arb(calls))
  IN yield(plugin(content[com[void]](init)),
                \\in. content[com[output]](apply(call, reflect(in))))
  ))
```

The `extern` function retrieves the structure named `ident` from the code repository. Currently, the code repository is just a collection of directories containing appropriate object files and intermediate compilation results. A more sophisticated access to the code repository still needs to be defined.

[8] `&` denotes the continuation operator of the `com` monad.

3. Semantic Foundation

We take a short look on the foundation of reflections in Opal. The algebraic foundation relies on the more general concept of theories that are planned to be incorporated into the forthcoming language version Opal-2. Theories in Opal-2 have a close relationship to type classes in Haskell, as we will also discuss. Operationally, reflections are based on a syntactic transformation.

3.1. Algebraic Foundation

In a setting such as Opal's which is based on concepts of algebraic specification (see for example [13]), declarations of the kind `ASSUME Dynamic[T]` can be understood as a *non-conservative enrichment* by a special kind of specification module. Following the naming convention used in OBJ for a similar construct [6] these modules are called *theories* as they do not represent executable program structures but assertions about them. In case of the theory `Dynamic`, it contains the assertion that a constant `reflectSort` exist:

```
THEORY Dynamic[alpha]
  SORT alpha
  FUN  reflectSort : sort'Reflection
```

In stating `ASSUME Dynamic[T]` in the context of a structure, we add to its name scope the constant `reflectSort'Dynamic[T]`. Though the type `T` is not part of the type of `reflectSort`, it is part of its name. Thus we have different constants `reflectSort'Dynamic[int]` (`reflectSort[int]` as an abbreviation), `reflectSort[bool]` and so on. The instances of `reflectSort` are exactly the basic information required to implement reflections, as we will see later.

The constant `reflectSort` introduced by `ASSUME` has no implementation, but it is ensured to exist. Hence `ASSUME` is, in contrast to `IMPORT`, not a conservative enrichment since it constrains the models of `T`. In the process of instantiating a parameterised structure, any assumptions it makes on its parameters are also instantiated and propagate

to the instantiation context. This way the assumptions accumulate upwards the import hierarchy of parameterised structures.

Assumptions are finally satisfied by providing definitions for the constants and functions. For example, in Opal-2 we state:

```
ASSUME Dynamic[int]
...
DEF reflectSort[int] == sort(name("int", sortKind, struct("Int", <>)))
```

For the restricted usage of theories and assumptions for adding reflections to Opal, the above definition can in fact be generated automatically by the compiler as soon as the type `T` is instantiated with a basic type in `ASSUME Dynamic[T]`. Since in every program all instantiations are eventually basic, we can always satisfy all `Dynamic` assumptions.

The general concept of assumptions and theories is nothing new – it is derived from Haskell's type classes, transferred to a setting of parameterised entities. This raises the question whether a similar approach for modeling dynamic type information is applicable in Haskell. We will discuss this question in the conclusion.

3.2. Syntactic Transformation

Though we have ensured that the assumptions on the existence of certain functions and constants are satisfied, it is not yet clear how they are communicated to their application points. This is achieved by adding them to the parameter list of structures, as in the example below:

```
SIGNATURE S[a, n, b]              SIGNATURE S[a, n, b, reflectSort_a]
  SORT a b                          SORT a b
  FUN n: nat              ==>       FUN n: nat
  ASSUME Dynamic[a]                 FUN reflectSort_a : sort'Reflection
```

As demonstrated in the next section, this transformation is sufficient for implementing the core reflection interface.

4. Implementation

This section discusses how the reflection interfaces introduced in the previous sections are implemented in the current Opal system. First, we discuss the implementation of *the construction of value reflections*. It is founded on the syntactic transformation introduced in the previous section. Second, we briefly discuss how *reflections of free types* and *function lookup and binding* are implemented.

4.1. Construction of Value Reflections

We now demonstrate how the core types depicted in Figure 1 can be implemented in conjunction with the syntactic transformation introduced in the previous section. We concentrate on the only difficult implementation, the type `value`. It is implemented as a pair consisting of a field `actual` storing some reflected object and a type tag field. The type tag is used to save the real type of the stored object. The field `actual` itself has type `SOME'RUNTIME` which is part of Opal's low-level runtime library and allows unsafe casting:

```
IMPLEMENTATION Reflection
  DATA value == pair (actual : SOME, type : type)
```

One way to construct such a `value` is to use the functions declared in the core reflection interface, see Figure 2. The functions declared in this interface can be implemented as follows, using the non-typesafe runtime system primitives `asSOME` and `fromSOME` in a typesafe way:

```
IMPLEMENTATION ReflectionBuild[alpha]
  SORT alpha
  ASSUME Dynamic[alpha]

  DEF reflect(x) == pair(asSOME(x),basic(reflectSort[alpha]))

  DEF reflects?(refl) ==
    IF basic?(type(refl)) THEN sort(type(refl)) = reflectSort[alpha]
                          ELSE false FI

  DEF content(refl) ==
    IF reflects?(refl) THEN fromSOME[alpha] (actual(refl)) FI
```

Recall from the previous section that the theory `Dynamic` declares a constant `reflectSort`. Furthermore, the compiler will automatically syntactically transform the above code into the following plain Opal structure:

```
IMPLEMENTATION ReflectionBuild[alpha, reflectSort_alpha]
  SORT alpha
  FUN reflectSort_alpha : sort

  DEF reflect(x) == pair(asSOME(x),basic(reflectSort_alpha))
  ...
```

4.2. Free Type and Function Reflection

The implementation of both the inspection interface, shown in Figure 3, as well as the lookup interface, shown in Figure 4, use a code repository. For every structure the repository stores a file describing the structure's signature alongside an object file. The repository is searched for functions or sort variants first by structure and then by name.

Once a structure has been requested by the reflection system, information about it is cached internally.

Once a name has been retrieved it can be bound using the function `bind`, yielding a `value`. The binding process in turn utilises the existing Opal runtime system which provides non-typesafe service functions for the dynamic linking of object files. Just like the core reflection interface, the implementation of the lookup interface effectively hides all unsafe casts and direct function calls from the user.

5. Conclusion

We have shown how generic meta programming can be integrated into an existing compilation system for a functional language. Reflections allow the programmer to query information about functions and types and to use that information in algorithms. The exemplary applications – virtual methods, generic printing and dynamically loadable plugins – give an impression of the possibilities generic meta programming offers.

A syntactic transformation provides a plain operational model of reflections. It has been implemented with only moderate extensions of the Opal compiler implementation. Indeed, the syntactic transformation and the code repository are all that is needed to provide the user with the full power of the reflection mechanism *while the compiler backend does not see reflections at all.*

In the current homogeneous approach of the Opal system, where several instances of a parameterised structure share the same code, reflections incur a runtime overhead. However, if an heterogeneous approach is taken, reflections come for free. Moreover, if this approach is combined with partial evaluation, reflection based algorithms allow for optimisations that would not be possible otherwise.

Future Research. Reflections as currently implemented in the Opal compiler provide new opportunities for generic programming. We have implemented several further libraries that could not be presented in this paper (for example generic implementations of relations and catamorphism), but the application potential still needs future research.

The forthcoming Opal-2 includes algebraic laws, theories and assumptions. In the Opal-2 implementation these should be subject to reflection as well as structures, functions and sorts. This allows new possibilities for the implementation of parameterised functions, because algebraic properties (other than the free type properties) can be exploited in the implementation of functions. Consider the reduce function on sequences which uses a tail-recursive implementation if the operator is associative. The following implementation works for both associative and non-associative functions:

```
FUN reduce: (alpha ** beta -> beta) -> beta ** seq[alpha] -> beta
DEF reduce(+) ==
  IF lookupAssumption("Associative", +) THEN reduceFast(+)
                                        ELSE reduceStd(+) FI

DEF reduceFast(+)(e, s) ==
  IF s empty? THEN e
  IF rt(s) empty? THEN ft(s) + e
              ELSE reduceFast(+)(e, ft(s) + ft(rt(s)) :: rt(rt(s))) FI
DEF reduceStd(+)(e, s) ==
  IF s empty? THEN e
              ELSE ft(s) + reduceStd(+)(e, rt(s)) FI
```

Such kinds of implementation become feasible only if partial evaluation is involved. A combination of unfolding and specialised simplification rules for reflection primitives for type and assumption information available at compile time, should result in the desired efficiency.

Related Work. In a functional setting, an alternative approach to dynamic types is described by Leroy and Abadi et al. [9,2]. Our approach differs in that it supports full instantiation information for parameters of polymorphic functions. In contrast, in the work of [9,2] a polymorphic function can only reflect the formal types, not the actual types of its parameters. As a consequence, in [9,2] no runtime type information needs to be passed as parameters to functions.

We regard it as a severe restriction of these systems that reflections of the instantiation types of polymorphic parameters cannot be obtained: for example, such reflections are needed to implement the "virtual method" construct. Most importantly, they allow to hide the use of reflections inside libraries.

In our model it is the syntactic transformation that adds the missing runtime type information as parameters to structures utilising reflections. A similar approach has

been taken by Pil [11] who extended the Clean compiler, such that full instantiation information is available for all functions that are marked by a special *type context*. This type context corresponds to the `ASSUME Dynamic` assumption and, indeed, the Clean compiler also performs a syntactic transformation of functions with type context similar to the transformation described above.

While the low-level integration of reflections into the Clean compiler core allowed the definition of a nice pattern-matching syntax for working with reflections, a distinct advantage of the more general approach using theories is that dynamic assumptions can be inferred *automatically* for structures and hence also for functions. Users can utilise libraries that employ reflections internally without having to bother about reflections at all, as long as the library declares `ASSUME Dynamic` somewhere. We believe that library programming is the actual realm of reflective programming.

It seems to have been known for some time in the Haskell community that dynamic types can be implemented in a similar way as described in this paper using type classes. However, a technical problem is that the following class definition is illegal:

```
class Dynamic a where
  reflectType :: Type
```

The member `reflectType` does not depend on `a` which is a required context condition in Haskell. One way to solve this problem is to add a dummy argument, yielding `reflectType :: a -> Type`. This technique has been used in the library of the Hugs-98 distribution for dynamics. Unfortunately it leads to "hacky" code, as for example in the following definition of a derived instance:

```
instance Dynamic a => Dynamic [a] where
  reflectType x = mkListType (reflectType (bottom x))
  where
    bottom :: [a] -> a
    bottom = bottom
```

The problem here is the way Haskell treats overloading. While Haskell exclusively uses type classes to resolve ambiguities, Opal (and C++) use ad-hoc overloading. In ad-hoc

overloading ambiguous partial names are promoted to complete names by inferring missing parts from context. While this solves the above problem, Opal's overloading mechanism could still benefit from an extension by Haskell's type classes: Type classes can be used to implicitly associate "internal" knowledge about data types with the specialised implementations of libraries that are parameterised over these types. We regard dynamic types and reflections as a typical application of this usage.

References

1. *The Opal Home Page*. http://uebb.cs.tu-berlin.de/~opal.

2. M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic Typing in Polymorphic Languages. *Journal of Functional Programming*, 5(1):111–130, Jan 1996.

3. Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic Typing in a Statically-Typed Language. In *16th ACM Symposium on Principles of Programming Languages*, pages 213–227, 1989.

4. Klaus Didrich, Andreas Fett, Carola Gerke, Wolfgang Grieskamp, and Peter Pepper. OPAL: Design and Implementation of an Algebraic Programming Language. In Jürg Gutknecht, editor, *Programming Languages and System Architectures, International Conference, Zurich, Switzerland, March 1994*, volume 782 of *Lecture Notes in Computer Science*, pages 228–244. Springer, 1994.

5. Klaus Didrich, Wolfgang Grieskamp, Christian Maeder, and Peter Pepper. Programming in the Large: the Algebraic-Functional Language Opal 2α. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages, St Andrews, Scotland, September 1997 (IFL'97), Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 323 – 338. Springer, 1998.

6. Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *12th ACM Symposium on Principles of Programming Languages*, 1985.

7. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. In *ESOP*, Jan 1994.

8. P. H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C. H. Flood, W. Grieskamp, J. H. G. van Groningen, K. Hammond, B. Hausman, M. Y. Ivory, R. E. Jones, J. Kamperman, P. Lee, X. Leroy, R. D. Lins, S. Loosemore, N. Röjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, P. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "pseudoknot", a Float-Intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

9. Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4):431–463, 1993.

10. Peter Pepper. *Funktionale Programmierung in* OPAL*,* ML*,* HASKELL *und* GOFER. Springer-Lehrbuch, 1998. ISBN 3-540-64541-1.

11. Marco Pil. Dynamic types and type dependent functions. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages, London, UK, September 1998, (IFL'98)*, volume 1595 of *Lecture Notes on Computer Science*, pages 169–185. Springer, 1999.

12. Sun Microsystems Inc. *JavaTM Core Reflection, API and Specification*, 1997. Part of the JDK documentation.

13. Martin Wirsing. *Handbook of Theoretical Computer Science*, chapter Algebraic Specification (13), pages 675–788. North-Holland, 1990. edited by J. van Leeuven.