Interfacing CDuce and OCaml

Giuseppe Castagna Julien Demouth Alain Frisch

Département d'Informatique
École Normalement Supérieure

PSD Lab, 2004-02-16

Outline

Outline

Outline

CDuce

CDuce is an XML-oriented functional and strongly typed functional language.

- Built on regular expression types and patterns (from XDuce)
- General purpose features: first-class/overloaded functions, ...

`http://cduce/`
`http://www.cduce.org/`

Makes OCaml available from CDuce

Scenario

A CDuce application that requires some OCaml code.

- Reuse existing libraries
  - Datastructures
  - Numerical computation
  - Bindings to C libraries: database, network
- Implement complex algorithms
- Ok to pass CDuce functions to OCaml higher-order functions (e.g.: iterators)

Makes OCaml available from CDuce

Scenario

A CDuce application that requires some OCaml code.

- Reuse existing libraries
  - Datastructures
  - Numerical computation
  - Bindings to C libraries: database, network
- Implement complex algorithms
- Ok to pass CDuce functions to OCaml higher-order functions (e.g.: iterators)

Makes OCaml available from CDuce

Scenario

A CDuce application that requires some OCaml code.

- Reuse existing libraries
  - Datastructures
  - Numerical computation
  - Bindings to C libraries: database, network
- Implement complex algorithms
- Ok to pass CDuce functions to OCaml higher-order functions (e.g.: iterators)

Makes OCaml available from CDuce

Scenario

A CDuce application that requires some OCaml code.

- Reuse existing libraries
  - Datastructures
  - Numerical computation
  - Bindings to C libraries: database, network
- Implement complex algorithms
- Ok to pass CDuce functions to OCaml higher-order functions (e.g.: iterators)

Makes CDuce available from OCaml

Scenario

An OCaml application that requires some CDuce code.

- CDuce used as an XML input/output/transformation layer
- Examples:
  - config file
  - XML serialization of internal data
  - XHTML presentation of results

Makes CDuce available from OCaml

Scenario

An OCaml application that requires some CDuce code.

- CDuce used as an XML input/output/transformation layer
- Examples:
  - config file
  - XML serialization of internal data
  - XHTML presentation of results

Makes CDuce available from OCaml

Scenario

An OCaml application that requires some CDuce code.

- CDuce used as an XML input/output/transformation layer
- Examples:
  - config file
  - XML serialization of internal data
  - XHTML presentation of results

The current situation

- It it possible to embed the CDuce interpreter in an OCaml application.

Drawbacks

- Explicit calls to the interpreter.
- All the CDuce values have a single common type `Value.t` as seen from OCaml.
- Need to write coercions by hand.
- Loose type safety.

The current situation

- It it possible to embed the CDuce interpreter in an OCaml application.

Drawbacks

- Explicit calls to the interpreter.
- All the CDuce values have a single common type `Value.t` as seen from OCaml.
- Need to write coercions by hand.
- Loose type safety.

The current situation

- It it possible to embed the CDuce interpreter in an OCaml application.

Drawbacks

- Explicit calls to the interpreter.
- All the CDuce values have a single common type `Value.t` as seen from OCaml.
- Need to write coercions by hand.
- Loose type safety.

The current situation

- It it possible to extend the CDuce interpreter with new "operators".

Drawbacks

- The procedure to extend the interpreter is heavy.
- Cannot just import an external library as a whole.
- Need to write coercions by hand.
- Loose type safety.

The current situation

- It it possible to extend the CDuce interpreter with new "operators".

Drawbacks

- The procedure to extend the interpreter is heavy.
- Cannot just import an external library as a whole.
- Need to write coercions by hand.
- Loose type safety.

The current situation

- It it possible to extend the CDuce interpreter with new "operators".

Drawbacks

- The procedure to extend the interpreter is heavy.
- Cannot just import an external library as a whole.
- Need to write coercions by hand.
- Loose type safety.

Users want more !

CDuce users mainly come from the OCaml community: they want to be able to reuse existing libraries that they know.

Example

A user wrote a set of new operators to write CGI scripts in CDuce.

Example

Another would like to use CDuce has a thin XML I/O layer, to specify how to map concrete XML to abstract internal representation.

Users want more !

CDuce users mainly come from the OCaml community: they want to be able to reuse existing libraries that they know.

Example

A user wrote a set of new operators to write CGI scripts in CDuce.

Example

Another would like to use CDuce has a thin XML I/O layer, to specify how to map concrete XML to abstract internal representation.

Users want more !

CDuce users mainly come from the OCaml community: they want to be able to reuse existing libraries that they know.

Example

A user wrote a set of new operators to write CGI scripts in CDuce.

Example

Another would like to use CDuce has a thin XML I/O layer, to specify how to map concrete XML to abstract internal representation.

The main challenge

To preserve type safety and avoid writing ad hoc coercions: need a mapping between OCaml types and CDuce types.

The good news: similar type constructors

Record types, product types, function types.

The bad news: different features

- OCaml: parametric polymorphism, abstract types, named types, objects, ...
- CDuce: overloaded functions, regular expressions, ...

The main challenge

To preserve type safety and avoid writing ad hoc coercions: need a mapping between OCaml types and CDuce types.

The good news: similar type constructors

Record types, product types, function types.

The bad news: different features

- OCaml: parametric polymorphism, abstract types, named types, objects, ...
- CDuce: overloaded functions, regular expressions, ...

The main challenge

To preserve type safety and avoid writing ad hoc coercions: need a mapping between OCaml types and CDuce types.

The good news: similar type constructors

Record types, product types, function types.

The bad news: different features

- OCaml: parametric polymorphism, abstract types, named types, objects, ...
- CDuce: overloaded functions, regular expressions, ...

The main design choice

One way mapping for types: OCaml $\Longrightarrow$ CDuce.

From an OCaml type t, deduce a CDuce type $\hat{t}$ and two coercion functions $t \leftrightarrow \hat{t}$.

An easy choice

Indeed, a mapping CDuce $\Longrightarrow$ OCaml wouln't be compatible with subtyping (except a uniform representation `Value.t`).

The main design choice

One way mapping for types: OCaml $\Longrightarrow$ CDuce.

From an OCaml type t, deduce a CDuce type $\hat{t}$ and two coercion functions $t \leftrightarrow \hat{t}$.

An easy choice

Indeed, a mapping CDuce $\Longrightarrow$ OCaml wouln't be compatible with subtyping (except a uniform representation `Value.t`).

Using a CDuce value from OCaml

- Let v be a CDuce value of type τ.
- To use it from OCaml: specify an OCaml type t
- Check that $\tau \leq \hat{t}$ and apply the coercion $\hat{t} \longrightarrow t$.

There might be several types t that are ok.

The burden of putting the value into the correct form is on the CDuce side (good for data transformation!).

Using an OCaml value from CDuce

- Let v be an OCaml value of type t.
- To use it from CDuce: apply the coercion $t \longrightarrow \hat{t}$.

The type of the result is uniquely determined (modulo subtyping).

Overview of the mapping

t	$\hat{t}$
`int`	`min_int` $--$ `max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\hat{t_1}, \hat{t_2})$
$t_1 \rightarrow t_2$	$\hat{t_1} \rightarrow \hat{t_2}$
t `list`	$[\hat{t}*]$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(`A_1, \hat{t_1}) \mid \ldots \mid (`A_n, \hat{t_n})$
$[`A_1$ `of` $t_1 \mid \ldots \mid `A_n$ `of` $t_n]$	$(`A_1, \hat{t_1}) \mid \ldots \mid (`A_n, \hat{t_n})$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{\|l_1 = t_1; \ldots; l_n = t_n\|\}$

Other OCaml features

Abstract types

Add OCaml abstract types to the CDuce type algebra.
Corresponding values are black boxes that expose only their type.

Polymorphic functions

Need to specify the instantiation before using a polymorphic OCaml function from CDuce (may often infer it).

Other OCaml features

Abstract types

Add OCaml abstract types to the CDuce type algebra.
Corresponding values are black boxes that expose only their type.

Polymorphic functions

Need to specify the instantiation before using a polymorphic OCaml function from CDuce (may often infer it).

Granularity

Don't mix OCaml and CDuce code

It is always messy to mix languages in a single source file (syntax highlighting, error localization, ...)

Using OCaml from CDuce

Just use the OCaml value!

E.g.: `Postgres.query c "select * from employees;"`

Granularity

Don't mix OCaml and CDuce code

It is always messy to mix languages in a single source file (syntax highlighting, error localization, ...)

Using OCaml from CDuce

Just use the OCaml value!
E.g.: `Postgres.query c "select * from employees;"`

Granularity

Unified vision: CDuce produces OCaml compilation units

The CDuce part integrates well into OCaml projects.

Using CDuce from OCaml

Import a whole CDuce unit at once by providing an OCaml interface (.cmi) for it.
E.g.:

```
a.cd:   let f(s : Latin1) : Latin1 = ...
a.mli:  val f : string -> string
b.ml:   let s = A.f "foobar"
```

Granularity

Unified vision: CDuce produces OCaml compilation units

The CDuce part integrates well into OCaml projects.

Using CDuce from OCaml

Import a whole CDuce unit at once by providing an OCaml interface (.cmi) for it.
E.g.:

```
a.cd:   let f(s : Latin1) : Latin1 = ...
a.mli:  val f : string -> string
b.ml:   let s = A.f "foobar"
```

Compiling a CDuce module in a mixed OCaml/CDuce project

1. Compile the OCaml interface for the CDuce module:
   `ocamlc -c a.mli` $\rightsquigarrow$ `a.cmi`
2. Compile the CDuce module:
   `cduce --ocaml a.cd` $\rightsquigarrow$ `a.ml, cduce_a.ml`
   (reads `a.cmi` and other compiled interfaces)
3. Compile the OCaml modules:
   `ocamlc -c cduce_a.ml a.ml` $\rightsquigarrow$ `cduce_a.cmo a.cmo`
4. Link:
   `ocamlc -o prog ... cduce_a.cmo a.cmo ...`

Compiling a CDuce module in a mixed OCaml/CDuce project

1. Compile the OCaml interface for the CDuce module:
   `ocamlc -c a.mli` $\rightsquigarrow$ `a.cmi`
2. Compile the CDuce module:
   `cduce --ocaml a.cd` $\rightsquigarrow$ `a.ml, cduce_a.ml`
   (reads `a.cmi` and other compiled interfaces)
3. Compile the OCaml modules:
   `ocamlc -c cduce_a.ml a.ml` $\rightsquigarrow$ `cduce_a.cmo a.cmo`
4. Link:
   `ocamlc -o prog ... cduce_a.cmo a.cmo ...`

Compiling a CDuce module in a mixed OCaml/CDuce project

1. Compile the OCaml interface for the CDuce module:
   `ocamlc -c a.mli` $\rightsquigarrow$ `a.cmi`
2. Compile the CDuce module:
   `cduce --ocaml a.cd` $\rightsquigarrow$ `a.ml, cduce_a.ml`
   (reads `a.cmi` and other compiled interfaces)
3. Compile the OCaml modules:
   `ocamlc -c cduce_a.ml a.ml` $\rightsquigarrow$ `cduce_a.cmo a.cmo`
4. Link:
   `ocamlc -o prog ... cduce_a.cmo a.cmo ...`

Compiling a CDuce module in a mixed OCaml/CDuce project

1. Compile the OCaml interface for the CDuce module:
   `ocamlc -c a.mli` $\rightsquigarrow$ `a.cmi`
2. Compile the CDuce module:
   `cduce --ocaml a.cd` $\rightsquigarrow$ `a.ml, cduce_a.ml`
   (reads `a.cmi` and other compiled interfaces)
3. Compile the OCaml modules:
   `ocamlc -c cduce_a.ml a.ml` $\rightsquigarrow$ `cduce_a.cmo a.cmo`
4. Link:
   `ocamlc -o prog ... cduce_a.cmo a.cmo ...`

Compiling a CDuce module in a mixed OCaml/CDuce project

Remarks

- `cduce_a.ml` contains the compiled version of `a.cd`; OCaml is used as a backend for CDuce.
- `a.ml` contains the stubs to expose the values of `cduce_a.ml` as typed OCaml values.
- Could as well use the OCaml native compiler.
- The CDuce compilers needs to read undocumented `.cmi` files.

Status of the implementation

Still under development!

- Mapping for types and coercions: almost done (todo: abstract types + polymorphism).
- Importing CDuce units from OCaml: done.
- Calling OCaml from CDuce: next step.
- Use OCaml as a backend for CDuce: todo (might live with an interpreter for some time).