Multi-polymorphic programming in bondi

Thomas Given-Wilson  
NICTA  
Thomas.Given-Wilson@nicta.com.au

Freeman Huang  
Queen’s University, Canada  
huang@cs.queensu.ca

Barry Jay  
University of Technology, Sydney  
Barry.Jay@uts.edu.au

Abstract
The bondi programming language is multi-polymorphic, in that it supports four polymorphic programming styles within a small core of computation, namely a typed pattern calculus. bondi’s expressive power is illustrated by considering the problem of assigning reviewers to a paper. As the context generalises from a committee to a committee with additional reviewers, to a conference, to a federation or confederation, the solution incorporates polymorphism familiar from the functional, generic functional, relational, path-based, and object-oriented programming styles, respectively. These experiments show that multi-polymorphic programming is both practical and desirable.

Keywords  polymorphism, pattern matching, bondi, pattern calculus, generic programming

1. Introduction
The strength of each programming style emerges from the uniform treatment of its central concepts: functional programming treats functions; relational programming treats relations; path programming treats semi-structured data, such as XML; object-oriented programming treats objects. However, it is not so easy to combine these concepts within a single system: making all of these concepts equally fundamental produces a chimera, which undermines uniformity, while middleware must ignore all such concepts.

This paper shows that these concepts of functions, relations, paths, and objects, can be freely expressed and combined in a small programming language, bondi [1] based on pattern calculus [7, 9], in which computation evaluates pattern-matching functions. Since the proof is too big for the margin of the proceedings, we illustrate the approach and ideas through a motivating problem, implemented in bondi and supported by a sketch of the underlying theory presented in detail in “Pattern Calculus: Computing with Functions and Data Structures” [7]. This problem is to find reviewers for a paper submitted to a conference. The interest comes from considering how, and where the reviewers are represented, and deciding whether or not they are suitable for a paper.

The solutions are presented in bondi. Generally speaking, programs consist of let-declarations and abstract data type declarations, in the ML-style [12], and class declarations in the style of Java [2] or C# [3]. New notation will be explained when it appears for the first time.

The simplest solution for the reviewer-matching problem is given by

\[
\text{filterList \ (okToRev \ paper) \ committee} \quad (1)
\]

in which committee is a list of Reviewer, which in turn is an abstract data type to represent a reviewer. \(okToRev: \text{Paper} \to \text{Reviewer} \to \text{Bool}\) determines if a reviewer is suitable to review a paper, and \(\text{filterList}\) of type

\[
(a \to \text{Bool}) \to \text{List} \ a \to \text{List} \ a
\]

is the usual operation for filtering a list by some predicate. In Solution (1) the type variable \(a\) will be instantiated to type Reviewer.

More generally, the potential reviewers need not be in a single list. For example, suppose that the program committee forms one list and the past reviewers form another such list. These could be combined into a pair of lists, a list of lists, or a tree, etc. called reviewers: 

\[
\text{okToRev} : \text{Paper} \to \text{Reviewer} \to \text{Bool}
\]

is a list of Reviewer, which in turn is a type variable representing a structure. In this case, a more general, structure polymorphic filter is required, given by \(\text{filter}\): 

\[
(a \to \text{Bool}) \to \text{List} \ a \to \text{List} \ a
\]

and the solution becomes

\[
\text{filter} \ (\text{okToRev} \ \text{paper}) \ \text{reviewers}. \quad (2)
\]

Solution (2) requires a dedicated data structure for holding reviewers, but in practice they will be scattered throughout a larger data structure representing the conference as a whole. Such a structure usually contains other types of data besides reviewers, such as conference name, papers, and events. Reviewers may also appear in different substructures such as committee member list and past reviewer repository. To traverse such a structure for reviewers, generalise the predicate okToRev\ paper which acts on reviewers only, to the function okIsRev\ paper: 

\[
a \to \text{List} \ \text{Reviewer}
\]

takes an argument \(x\) of any type \(a\). If it is a suitable reviewer then the result is the singleton list containing \(x\) else the result is empty. Now the solution becomes

\[
\text{select} \ (\text{okIsRev} \ \text{paper}) \ \text{conference} \quad (3)
\]

where select has type

\[
(\text{all} \ a, a \to \text{Maybe} \ b) \to c \to \text{List} \ b.
\]

In our example, the type variable \(b\) is instantiated to the type of reviewers and \(c\) to that of the conference. The argument type \(\text{all} \ a, a \to \text{Maybe} \ b\) contains the quantified type variable \(a\) since okIsRev\ paper must be able to act on arguments of any type.

Now suppose that two conferences decide to federate, as represented by some data structure confa. Further, they agree to share reviewer data, so that a reviewer known to one conference can be

---

1 NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright © ACM [to be supplied]. . . $15.00
used by the other. However, this sharing is not to apply to members of the respective program committees, who already have enough to do.

The solution requires awareness of where the different lists of reviewers sit within the structure of the conference federation, so that it has the flexibility to choose lists according to their role within the conference. For this purpose paths are ideally suited since they combine properties of the goals (e.g., reviewers) with properties of their location within the structure, as will be represented by stages. The solution presented is a variation of signposts [5], or, more generally, a pattern structure [6]. The signpost employed is

\[
\text{Stage (getRevs paper) (Goal (okIsRev paper))}
\]

where Stage (getRevs paper) finds a list containing reviewers according to their role (relative to the given paper) and Goal (okIsRev paper) chooses suitable reviewers as before. Then the overall solution becomes

\[
\text{navigate \ (Stage(getRevs paper) \ (Goal(okIsRev paper))) \ confs}
\]

(4)

where navigate: SignPost a -> b -> List a is the pattern-matching function that navigates through the stages, to the goal.

If the two conferences are willing to create a central administration holding all their data then the above solution is fine. However, a more likely scenario is that they will choose a confederation, in which each conference maintains slightly different information about its reviewers. This can be handled using object-oriented classes. Create a super-class of reviewers, with sub-classes for specialised reviewers as required. Now getRevs and okIsRev must be adapted to handle classes, and the structure of the conference is also adapted. The object-oriented solution is thus

\[
\text{navigate \ (Stage(getRevs2 paper) \ (Goal(okIsRev2 paper))) \ confs}
\]

(5)

Note that although getRevs2 and okIsRev2 are given as functions, they can be defined using methods which have special cases for different classes of reviewers.

These solutions illustrate four different sorts of polymorphism: the data polymorphism of filterList, the structure polymorphism of filter, the path polymorphism of select and of navigate, and the inclusion polymorphism of the reviewer classes.

While fragments of the expressive power displayed here can be found in other settings, no other calculus or typed programming language is able to support such breadth. For example: the structural simplicity, we begin by assuming that all reviewers have the necessary expertise, so that the challenge is to exclude reviewers who have a conflict of interest, either because they are an author of the paper in question, or a close associate of such an author.

In this section, the potential reviewers are exactly the members of the program committee, who are represented by a list, so that the solution is simply to filter the list in the usual functional style.

The first polymorphic list function required is the isMember function

\[
\text{let rec \ (isMember : a -> List b -> Bool) \ x =}
\]

\[
\text{| Nil \rightarrow False}
\]

\[
\text{| Cons \ y \ ys \rightarrow equal(x,y) || (isMember \ x \ ys)}.
\]

(6)

which checks for equality of its first argument with any entry of its list argument. isMember makes use of bondi’s generic equality

\[
\text{equal: a \cdot b \rightarrow Bool}
\]

(as defined in Section ??) that is able to compare arbitrary terms. Type safety can be strengthened by restricting the type to a \cdot a \rightarrow Bool but later, more polymorphic examples, will require the greater flexibility of the generic equal.

Much of the notation should be familiar from ML and other functional programming languages: let rec introduces recursion; the vertical bar \mid \text{separates cases of a pattern-matching function;} \text{Nil} and Cons are the usual list constructors; and the double bar \mid \text{disjuncts.}

The second polymorphic list function is

\[
\text{disjoint: List a \rightarrow List b \rightarrow Bool}
\]

that is True if its arguments have no members in common. This can be easily implemented using the isMember function. The third polymorphic list function employed is

\[
\text{filterList: a \rightarrow Bool) \rightarrow List a \rightarrow List a}
\]

Its first argument is a boolean function that serves as a predicate to filter the list members given by the second argument. The implementation is straightforward; the predicate is applied to list members one by one, and only those satisfying the predicate stay.

The specifics of the reviewing problem require the following abstract data types representing people, reviewers and papers, respectively.

- **datatype Person** = Name of String
- **datatype Reviewer** = Rev of Person and List Person
- **datatype Paper** = Paper of String and List Person

A reviewer Rev person assocs is given by the person and a list assocs of their associates. A paper Paper title auths provides a title title and a list auths of authors.

Conflicts are avoided using the function okToRev given by

\[
\text{let okToRev : Paper → Reviewer → Bool} =
\]

\[
\text{| Paper title auths →}
\]

\[
\text{| Rev person assocs → disjoint (Cons person person assocs) auths}
\]

Reviewer conflicts of interest are avoided using the function okToRev that is True if the authors of a paper and associates of a reviewer are disjoint. Thus the complete Solution (1) is given by

\[
\text{let findReviewers paper = filterList (okToRev paper)}.
\]

A concrete example arises from considering the paper paper defined by

\[
\text{Paper "Calculus Revisited"}
\]

\[
\text{[Name "B",Name "C",Name "N"]}
\]

and two reviewers

2. Data polymorphism

Our running problem is to assign reviewers to a paper. For simplicity, we begin by assuming that all reviewers have the necessary
rev\text{A} = \text{Rev} (\text{Name "A"}) [\text{Name "H"}]
and
rev\text{B} = \text{Rev} (\text{Name "B"}) [\text{Name "I"}, \text{Name "J"}].

Now \text{findReviewers} paper \{\text{revA, revB}\} evaluates to (the value of) \{\text{revA}\}.
\text{bondi} supports an interactive environment. Given the declarations above then the evaluation of \text{findReviewers} produces the following display:

\text{let okRevs = findReviewers paper \{\text{revA, revB}\};;}
\text{okRevs: List Reviewer}
\text{okRevs = \{ Rev (Name "A") [Name "H"] \}}

The " is the \text{bondi} prompt and input is terminated by ;;. The system reports the type and value of the resulting list.

3. Structure polymorphism

The limitation of the solution above is that the potential reviewers must be represented by a single list. In practice, it will often happen that reviewers are drawn from varying structures, containing past authors and reviewers, as well as the program committee. The challenge is thus to work with an arbitrary collection of reviewers, without first converting it into a list.

This is achieved by generalising the function \text{filterList} to a more generic
\text{let filter \text{pred} =}
\text{foldleft (fun \text{x y} ->}
\text{if (\text{pred \text{y}})
\text{then Cons \text{y} \text{x}}
\text{else \text{x})}
\text{[]);}

that constructs a list of results that are True for the predicate \text{pred}.

While the program committee may be stored in a list, a larger pool of reviewers may be kept in a more efficient structure such as a tree. To represent a sample of such data structures the following data types are defined
\text{datatype PairList \text{a} =}
\text{PairList of List \text{a} and List \text{a}}
\text{datatype Tree \text{a} =}
\text{Leaf of \text{a} | Node of Tree \text{a} and Tree \text{a}}
\text{datatype ListTree \text{a} =}
\text{ListTree of List \text{a} and Tree \text{a}.}

The PairList \text{a} is simply a pair of lists containing elements of type \text{a}. The Tree is a simple representation of a tree style data structure. The ListTree \text{a} combines both a list of \text{a}’s and a Tree of \text{a}’s. The general Solution (2) for filtering an arbitrary data structure of reviewers is now given by

\text{let findReviewers paper = filter (okToRev paper).}

Continuing from the previous example, define two new reviewers
\text{rev\text{C} = Rev (Name "C") [Name "K"]}
\text{and}
\text{rev\text{D} = Rev (Name "D") [Name "L"].}

Then both
\text{findReviewers paper}
\text{(PairList [rev\text{A}, rev\text{B}] [rev\text{C}, rev\text{D}])}
\text{and}
\text{findReviewers paper}
\text{(ListTree [rev\text{A}, rev\text{B}] (Leaf rev\text{D}))}

evaluate to \{\text{rev\text{D}, rev\text{A}}\} as shown here.

\text{let okRevs1 = findReviewers paper (PairList [rev\text{A}, rev\text{B}] [rev\text{C}, rev\text{D}]);;}
\text{okRevs1: List Reviewer}
\text{okRevs1 = \{ Rev (Name "D") ([Name "L"])
Rev (Name "A") ([Name "H"]) \}}

\text{let okRevs2 = findReviewers paper (ListTree [rev\text{A}, rev\text{B}] (Node (Leaf rev\text{C}) (Leaf rev\text{D})));;}
\text{okRevs2: List Reviewer}
\text{okRevs2 = \{ Rev (Name "D") ([Name "L"])
Rev (Name "A") ([Name "H"]) \}}

\text{okRevs1 \text{== okRevs2};;}
\text{it: Bool}
\text{it = True}

Observe that the underlying structure is immaterial to the results as shown by comparing okRevs1 and okRevs2.

4. Path polymorphism

A limitation of the structure polymorphic approach is that it requires a clear separation of the data of interest from the structure that contains it. However, a conference typically contains several sorts of data, concerning papers, authors, a program committee, a repository of past reviewers, etc. in which case it is infeasible to represent the data as a structure parameterised by reviewers. Now the challenge is to find reviewers within a larger conference structure with many types of elements and substructures such as the following:

\text{datatype Committee =}
\text{Comm of List Reviewer}
\text{datatype Reviewers =}
\text{Revs of List Reviewer}
\text{datatype Conference =}
\text{Conf of String and List Paper and Committee and Reviewers}

In practice, the structure may be more complicated, and vary between conferences or over time.

To traverse a data structure without knowledge of the components requires a function that is path polymorphic. It should be able to walk through different paths to reach target elements that are located in different places, and be able to handle other types of elements encountered en route. Performing such a traversal and selecting relevant elements to collect can be done by the \text{select} function with the type

\text{(all \text{a.a} -> List \text{b}) -> c -> List \text{b}}

where the \text{a} must be for all types, \text{b} is the type of the desired results and \text{c} is the type of the data structure. Its definition is approximated by
let rec (select: (all a . a -> List b) -> c -> List b) =
  fun f ->
    | Ref (Ref _) -> []
    | Ref z -> select f z
    | z y -> append (f (z y)) (append (select f z) (select f y))
    | y -> append (f y) []

    ;;

where append is the usual list append. The first two cases that include the constructor Ref are exploited in the inclusion polymorphism of Section 6. The select function cannot be applied to okToRev for typing reasons, so the latter must be generalised to

let (okIsRev : Paper -> a -> List List Reviewer) =
  fun f ->
    if (okToRev paper (Rev name auths))
    then [Rev name auths]
    else []
    | z y -> append (f (z y)) (append (select f z) (select f y))
    | y -> append (f y) []

Now findReviewers paper = select (okIsRev paper).

An example conf1 of a conference can be defined as

Conf "Beta Programming" [paper]
  (Comm[revA,revB]) (Revs[revC,revD]).

Now findReviewers paper conf1 evaluates to [revA,revD].

5. Path matching

A limitation of the previous path polymorphic approach is that all reviewers in the conference are treated the same way, regardless of their role, but in practice it may be necessary to distinguish, say, committee members from other reviewers.

For example, suppose that two conferences are federated and agree to share reviewer lists, but not to ask members of the one conference to review papers for the other conference. The necessary path descriptions can be given as

```
let rec (navigate : SignPost b -> a -> List b) =
  fun z -> (navigate sign (select f z)).
```

For our conferences, the lists of reviewers are produced by a function

```
let (getRevs : Paper -> a -> List List Reviewer)
  paper =
    let (getRevs : Paper -> a -> List List Reviewer) =
      fun z -> (navigate sign (select f z)).
```

A concrete example is given involving two new reviewers revE and revF

revE = Rev (Name "E") [Name "M"]
revF = Rev (Name "F") []

and by defining a conference conf2

Conf "Information Theory" []
  (Comm [revE]) (Revs [revF]).

Then the evaluation of findReviewers proceeds as shown

```
let okRevs = findReviewers paper (conf1,conf2);
```

yielding three reviews, two from the first conference and one from the second. Observe that reviewer revF has been avoided due to being on the program committee of conf2 (and not due to conflicts).

6. Inclusion polymorphism

Although the solution above is able to support various sorts of conference organisation it nevertheless requires that types such as Reviewer be fixed in advance. This requirement can be relaxed by introducing an object-oriented class of reviewers and allowing conference organisers to declare their own reviewer sub-classes. Such an approach also allows the suitability of reviewers to be expressed by a method which can be specialised for different sorts of reviewers.

This section creates a class of reviewers and a single sub-class of specialists with areas of expertise used in deciding suitability. Some of the earlier data types must now be reworked as classes. In particular, corresponding to the type Person is the class PersonC

```
class PersonC {
  name : String;
  get_name = { | () -> !this.name };
  set_name = { fun n -> this.name = n }.
}
```

It has one field name of type String and two standard methods, the get and set methods for the name. This declaration introduces a new abstract data type with constructor

```
PersonC: a -> ref String -> PersonC[a].
```
whose first argument (of type a) represents any additional fields
the person may have, and whose second argument is an (assignable)
reference to a name string. A PaperC class is defined similarly with
fields title and authors for its fields.

Now reviewers are represented using a sub-class ReviewerC of
PersonC declared by

    class ReviewerC extends PersonC {
        associates : List (PersonC [Top]);
        get_associates = { | () -> this.associates }
        set_associates = { fun a -> this.associates = a }
        suitable_for = { fun paper ->
            disjoint (paper.get_authors())
            (Cons (this:PersonC[Top]) !this.associates)
        }
    }

Since the associates can be any sort of person, the field associates
has type List (PersonC [Top]) where Top is the top type, a
super-type of every type. The only non-trivial method in the class
is suitable_for of type PaperC[b] -> Bool which is de-
defined using the higher-order parametrically polymorphic function
disjoint.

The sub-class declaration introduces another constructor
ReviewerC : a -> ref (List PersonC[Top]) ->
ReviewerC a
to collect the new fields. These are combined with the fields of
the PersonC class so that the pattern for a reviewer is given by

    ReviewerC (ReviewerC rest assocs) name : PersonC

where rest is used to represent any additional fields associated
with further sub-classes. As the depth of the sub-class hierarchy
increases, so too will the size of the pattern and its type. To avoid
this, we introduce the syntax ReviewerC[a] as syntax for the pattern
PersonC (ReviewerC _) _ where _ matches anything.

The data types for the conference and substructures are recre-
ated to use the classes for paper and reviewers.

datatype Committee2 =
    Comm2 of List ReviewerC[Top]
datatype Reviewers2 =
    Revs2 of List ReviewerC[Top]
datatype Conference2 =
    Conf2 of String and List PaperC[Top] and
    Committee2 and Reviewers2

The getRev2 function can be defined similarly to getRev with
the new data types and classes. The same needs to be done for the
okIsRev function, only now the matching must be done as a
shown here

    let okIsRev2: PaperC[b] -> a -> List a) paper =
        | (x: ReviewerC[c]) ->
            if x.suitable_for(paper) then [x]
            else []
        | z -> [].

The object-oriented solution (5) is thus given by

    let navigate (Stage (getRev2 paper)
            (Goal (okIsRev2 paper)))

Now suppose that a conference wishes to refine the representa-
tion of papers, reviewers and their suitability. Assume a sub-class
KeywordPaperC of papers that have a field keywords representing
relevant areas of research.

A sub-class of specialists is defined by

    class SpecialistC extends ReviewerC {
        areas : List String;
        get_areas = { | () -> this.areas }
        set_areas = { fun a -> this.areas = a }
        suitable_for = {
            | (x:KeywordPaperC[b]) ->
                super.suitable_for(x) &&
                (not (disjoint x.get_keywords())) (!this.areas))
            | (x:PaperC[c]) -> False
        }
    }

A specialist is a reviewer who has areas of expertise, represented by
a list areas with its associated get and set methods. The method
suitable_for is specialised to first check that it is suitable as before, represented by the boolean super.suitable_for(...)
and also that there is at least one keyword among the areas of
expertise, as represented by

    (not (disjoint kp.get_keywords())) (!this.areas)).

The default case | (z:PaperC[c]) -> False expresses the in-
tention that specialist reviewers refuse to review papers that do not
specify keywords.

These additions to the class hierarchy require no modifica-
tions to the function findReviewers2, demonstrating dynamic
dispatch. More precisely, evaluation of findReviewers2 will in-
voke, for each entry in a list of type ReviewerC[Top], the appro-
priate specialisation of the method suitable_for.

For illustration, define revOA,revOB,paperC etc. to repre-
sent the same data as revA,revB,paper in Section 5 but using
the classes corresponding to the older data types. Also con-
sider a keyword paper kpaperC whose keywords are given by

    ["types", "calculus"]

A specialist speCG whose areas are given by

    ["formalism", "types"]

    let confs2 =
        (Conf2 "Beta Programming" [paperC]
            (Comm2 [revOA,revOB])
            , Conf2 "Information Theory" [kpaperC]
                (Comm2 [revCE])
                (Revs2 [(revCF:ReviewerC[Top])
                    , (speCG:ReviewerC[Top])])).

Evaluating findReviewers2 paperC confs2 produces

    [revCD,revCA,revCF]

since the specialist speCG is not interested in papers without key-
words. By contrast, findReviewers2 paperC confs2 evaluated to
[revCC,revCD,revCF, speCG,revCE], avoiding only the
committee members of the conference kpaperC does not belong to.

This is the last of our solutions to the reviewing problem. It
employs data polymorphism, path polymorphism, path matching
and inclusion polymorphism in a natural manner.

7. Conclusions and future work

The difficulty of combining programming styles is well known.
Perhaps the best example is the struggle to combine functional
and object-oriented programming, but similar issues arise in combin-
ing objects with relational data, or more recently in trying to de-
sign web-services. This paper offers a new approach to the issues,
through pattern-matching. The various styles are combined within
solutions to the simple, but telling, reviewing problem, of searching
unknown data structures for reviewers having the right roles,
whose suitability is determined dynamically.
Future work may consider even greater generality by supporting confederations in which there are no shared super-classes of reviewers. Then the goal is to dynamically create the patterns necessary to interact with unfamiliar data structures, by passing ontologies as parameters. The basic principles are illustrated in the

Another question raised by this approach is its relation to logic. The Curry-Howard Isomorphism identifies computations with proofs, and types with propositions, as exploited in, say, [2]. This suggests that the new forms of polymorphism should correspond to new proof principals but it is not clear what they should be. One possibility is that path polymorphism corresponds to proof by structural induction, a generalisation of induction over the natural numbers. Generalising the Peano principal leads to the following (informal) proof rule

\[
\forall c. P(c) \quad \forall x, y. P(x) \land P(y) \Rightarrow P(x \cdot y)
\]

in which \(c\) represents a constructor and \(x \cdot y\) is a compound data structure. Some closer ties between pattern calculus and logic have been explored by examining the expressiveness of combinatory logic [8], however further relation to bondi and programming languages remains.

The examples in this paper demonstrate the desirability of multi-polymorphic programming. The implementations in bondi show that multi-polymorphic programming is practical, in the sense that the expected behaviour has been realised in a small programming language whose syntax is relatively familiar.

That done, one common question is whether the core of pattern calculus is fundamental to such a language. Each programming style tries to balance an emphasis on functions with an emphasis on data structures. At the extremes are pure \(\lambda\)-calculus, in which everything is a function, and Turing machines, in which everything is a structure on tape. In between are relational languages, with some choices of structure and functionality, and object-orientation, which try to package the two together. From this viewpoint, the challenge is to combine functions and data structures without strain. Pattern-matching functions provide a perfect combination of structure, as described by patterns, and functionality, as described by the bindings. On this view, the goal is to make pattern-matching as expressive as possible. Indeed, the intensionality afforded by considering the structure of an argument to a function proves to yield greater expressive power than the \(\lambda\)-calculus [8].

Putting aside such theoretical concerns, even the most skeptical reader can accept that the reviewing problem poses a challenge which cannot be met from within any previous programming style. Yet the issues it raises are typical of the challenges arising when working with large and evolving systems full of people, products and processes. Indeed, they are central to our ability to engage with systems about which we have only partial information, as arise in web services. The implementation of the typed pattern calculus in bondi is a step towards the general solution of such problems.

References