Proc. Estonian Acad. Sci. Engng., 1998, **4**, 2, 119–129
https://doi.org/10.3176/eng.1998.2.04

ON THE BORDER BETWEEN FUNCTIONAL PROGRAMMING AND PROGRAM SYNTHESIS

Enn TYUGU

Royal Institute of Technology, Electrum 204, 164 40 Kista, Stockholm, Sweden;
e-mail: tyugu@it.kth.se

Received 19 January 1998

Abstract. The importance of compositionality in program construction is being accepted quite well. In this respect, relational programming has clear advantages over functional programming. Unfortunately, there is no general technique of relational programming efficient enough to compete with the existing functional programming techniques. Here we discuss structural synthesis of programs – a method of synthesis of functional programs explainable in terms of higher-order functional constraint nets, simple types or intuitionistic logic. This method has been used in the implementation of declarative languages that allow us to specify concepts as relations and use them in specifications more flexibly than functions.

Key words: functional and relational programming, structural synthesis, incremental development, partial deduction.

1. STRUCTURAL SYNTHESIS OF PROGRAMS

Structural synthesis of programs (SSP), known for a number of years, has been used at least in two commercial systems: XpertPriz and PRIZ [1]. It has been applied and extended in several ways. Here we summarize the recent extensions and refer to [2] for some earlier extensions.

SSP is a deductive program synthesis method based on the idea that we can construct programs taking into account only their structural properties. We use this idea for constructing programs from small as well as large modules whose behaviour we do not describe in detail. Each preprogrammed module is supplied with a specification used as an axiom, stating under which conditions it can be applied and which values it computes. However, the specification does not specify the relation between the input and output values. Instead of the general form of a program module specification $\forall x.(P(x) \to \exists y.R(x,y))$ with precondition P and postcondition R, we use $\forall x.(P(x) \to \exists y.R(y))$, which tells us that a correct value (satisfying R) can be computed, but does not show the relation between this value and the input. The latter formula has the following equivalent form $\exists x.P(x) \to \exists y.R(y)$. As a consequence, we can present axioms in a propositional language, considering the closed formulae $\exists x.P(x)$ and $\exists y.R(y)$ as propositions whose internal structure is inessential. This allows us to use logic in which we are able to handle theories (specifications for the synthesis of programs) with large amount (up to tens of thousands) of axioms.

The SSP uses an implicative fragment of the intuitionistic propositional calculus (IPC) with restricted nestedness of implications. Intuitionistic logic guarantees simplicity of program extraction from proofs: programs are realizations of formulae and can be represented in typed lambda calculus. The general form of formulae is

$$(\underline{U} \to V) \to (\underline{X} \to Y),$$

where, for any symbol W, $\underline{W}$ denotes $W_1 \& W_2 \& ... \& W_k$ or an empty formula. Propositional letters U, V, X, Y denote computability of objects. Hence, an implication $\underline{X} \to Y$ has the following meaning "y is computable if $x_1, x_2, ..., x_m$ are computable", where $y, x_1, x_2, ..., x_m$ are the variables whose computability is denoted by the propositions $Y, X_1, X_2, ..., X_m$. An implication of this form can be either a specification of a preprogrammed module (i.e., an axiom) or a goal specifying the program which has to be synthesized.

The nested implications $\underline{U} \to V$ in the formula $(\underline{U} \to V) \to (\underline{X} \to Y)$ are called subtasks, and they provide the required generality to the language. They play the same role in composing an algorithm as atoms do in the body of a Prolog clause – they state subgoals to be achieved for applying an axiom (a clause in Prolog). Program development based on SSP is sometimes called propositional logic programming, because first, it is a kind of logic programming and, second, it uses a propositional logic [3].

The fragment of IPC used in SSP is still expressive enough for deductively equivalent encoding of arbitrary IPC formulae [3]. The derivability problem in IPC is PSPACE complete, consequently, the proof-search in SSP in general is PSPACE complete, but efficient algorithms exist for practical cases with less complexity [4].

2. SEMANTICS OF SPECIFICATIONS IN NUT

It is inconvenient to use the logical language of SSP directly for writing specifications for program synthesis. Instead, this language has been applied for representing semantics of several high-level specification languages [1]. One of these is the NUT specification language considered here.

The NUT specification language is an object-oriented language extended with features for program synthesis. A class is specified as a collection of declarations of the following form:

Superclass declaration

`super C` – `C` is a class

Component declaration

`var A : C` – `A` is a new component, `C` is a class

Method declaration

`rel R :` *<axiom> <program>* – `R` is a new method name

Equivalence declaration

`rel A = B` – `A, B` are components

Equation declaration

`rel` *E*1 = *E*2 – *E*1, *E*2 are arithmetic expressions

Here we give examples of classes `set`, `subset` `exists` and `intersection` written in the NUT specification language. The keywords `num` and `bool` denote built-in classes for primitive data types, and the keyword `any` denotes a universal class which has to be narrowed to a known class before the synthesis starts. The programs which are realizations of methods are presented here by their names *get, f* and *g*.

```
class set
 var
   val : any;
   sel : num;
   elem : num;
 rel
   val & sel -> elem (get) %this relations produces elements of
                           %the set one by one for properly given
                           %values of the selector sel.

class subset
 var
   of : set;
   is : set;
   cond : bool;
 rel
   (of.sel -> cond) -> (of.val -> is.val) (f)
```

```
class exists
 var
   res : bool;
   in : set;
   cond : bool;
 rel
   (in.sel -> cond) -> (in.val -> resb) (g)

class intersection
  var
   A : set;
   B : set;
   res : set;
   P : exists;
   Q : subset;
 rel
   B = P.in; A = Q.of; P.res = Q.cond; res = Q.is;
   r  : A.val & B.val -> res.val (spec)
```

Any class C in NUT can be used as a specification for synthesis of a program which realizes a goal of the form

$$A\&...\&B \to D,$$

where *A*, *B*, ..., *D* are propositions stating the computability of components of the class C. The synthesis succeeds if the goal can be derived from the axioms of the class specification. An example of the goal for synthesis is in the last line of the class `intersection`:

```
 r  : A.val & B.val -> res.val (spec)
```

The keyword `spec` denotes that the realization of the axiom has to be synthesized, using other axioms of the class.

The following is a brief presentation of the logical semantics of specifications in NUT developed by Uustalu [5]. This semantics is given in a first-order language, where an important role belongs to classifying predicates telling to which class an object belongs. Each class `C` has an associated classifying predicate, which we denote by the classname itself so that *C*(*w*) states "an object *w* belongs to the class `C`". We denote a subtask by *S*. Underlining in the first formula denotes a union, in other formulae a conjunction as before. The semantic function *Sem* for translating specifications into the set of (first-order) axioms is defined as follows.

$Sem(\texttt{class C : } \underline{D}) =_d \forall w.C(w) \to \underline{Sem'(D, w)}$

– semantics of a class is a set of axioms obtained as a union of axioms of its declarations.

Sem´(`super C, w`) $=_d C(w)$

– axioms of a superclass are taken without changes.

Sem´(`var A : C, w`) $=_d C(w.A)$

– a component A of an object w obtains the name *w.A* and keeps the class *C* given in its declaration.

Sem´(`rel R :` $\underline{\mathtt{S}}$ $\underline{\mathtt{X}}$ `-> Y <program>, w`) $=_d \underline{Sem''(S,w)} \rightarrow Sem'(\underline{X} \rightarrow Y, w)$

– semantics of a method declaration is given by its axiom.

Sem´´($\underline{\mathtt{U}}$ `-> V, w`) $=_d \underline{computable(w.U)} \rightarrow computable(w.V)$

– semantics of a simple implication is a statement about computability of the object on its right side if the objects on its left side are computable.

Sem´(`rel X = Y, w`) $=_d w.X = w.Y$

– equality of components is encoded in the proper way.

An equation can be considered as an abbreviation for denoting the methods which are solving functions of the equation. Therefore, the semantics of an equation is expressible as the set of axioms for these methods. More precisely, semantics of an equation $\mathtt{E}_1 = \mathtt{E}_2$, which binds the variables $x_1, x_2, ..., x_k$ and can be uniquely solved for the variables $x_1, x_2, ..., x_m$, $m \leq k$, is the following:

$$Sem'(\mathtt{E}_1 = \mathtt{E}_2) =_d \underline{x_1 \& \ldots x_{i-1} \& x_{i+1} \& \ldots x_k} \rightarrow x_i$$

with an axiom for every x_i, $i = 1, ..., m$.

Inference rules for transforming specifications into the language of SSP are the following (notation $[x]$ denotes an occurrence of x in the respective formula, t, t_1, t_2 are terms):

$$\frac{\forall w.C(w) \rightarrow D[w] \qquad C[t]}{D[t]}$$

$$\frac{t_1 = t_2 \qquad R[t_1]}{R[t_2]} \qquad\qquad \frac{t_1 = t_2 \qquad R[t_2]}{R[t_1]}$$

The proof-search is organized in such a way that these rules are applied only at the beginning of a proof. This results in unfolding of the set of axioms so that axioms instantiated for each object are explicitly included into the unfolded theory. After unfolding of the set of axioms, one can work in the propositional language of SSP.

3. SEMANTICS OF PRODUCTION RULES

Attempts have been made to extend the restricted logic of NUT by introducing production rules. Semantics of rules in NUT has been recently investigated in [6]. The general form of a production rule in NUT is

`rule` P_1`&. . .&`P_k `->` Q_1`&. . .&`Q_m ($\underline{P}$ `->` $\underline{Q}$ in the abbreviated form),

where P_i, Q_j are atoms of the form

```
T : C A1 = T1,...,An = Tn,
```

denoting that an object `T` belongs to the class `C` and its components `A1,...,An` are bound by the equivalences `A1 = T1,...,An = Tn`. (`T` is an object variable, `T1,...,Tn` may be constants or object variables beginning with the symbol ¤.)

Example:

```
rule
¤a:Point & ¤b:Point -> ¤c:Bar p= ¤a, q = ¤b
```

Semantics of rules is given by means of a function *Rsem*, which translates the rules into geometrical clauses in intuitionistic logic – into the first-order formulae of the form $\forall w.\ (F \to G)$, where *F* and *G* are built of atomic formulae using only conjunction, disjunction, falsity constant, and the existential quantifier [7].

$$Rsem(\texttt{rule}\ \underline{P}\ \texttt{->}\ \underline{Q}) =_d \forall w.(\underline{K(P)})\ \texttt{->} \exists z.\underline{K(Q)})$$

$$K(T : C\ A = T') =_d C(T)\ \&\ \underline{T.A = T'}$$

the variables $\underline{w}$ appear in $\underline{P}$, $\underline{z}$ appears in $\underline{Q}$ and not in $\underline{P}$; *T*, *T′* are object terms (i.e., a variable denoting an object or a composite name with such a variable in it).

In geometric theories, only atomic formulae play a critical role, and they can be handled by natural deduction rules in a certain simple form. The extra inference rule needed for the rules part of NUT is the following admissible inference rule of the geometric logic:

$$\frac{\forall\ \underline{w}.\ P[\underline{w}] \to Q[\underline{w}, \underline{z}] \qquad P[\underline{t}] \qquad \begin{array}{c} Q[\underline{t}, \underline{z}'] \\ \vdots \\ R[\underline{t}] \end{array}}{R[\underline{t}]},$$

where $\underline{z}'$ are fresh variables, and notation [*x*] denotes occurrence of *x* in respective formulae.

4. ANNOTATIONS OF AXIOMS

In order to specify building blocks of synthesized programs more precisely than the SSP language permits, one should use a richer language for annotating axioms. These annotations can be used both for verification of synthesized programs and for guidance of the proofs-search in order to avoid incorrect solutions. Tammet has proved correctness of some set manipulation programs [8], using first-order annotations of axioms of SSP. The idea of annotations can be explained on the following simple example. Let us have the following three axioms with realizations and annotations

$$\begin{array}{lll} x \to y & (\lambda x.\, f) & \forall x\, .f(x) = \sin(x) \\ x \to z & (\lambda x.\, g) & \forall x\, g(x) = \cos(x) \\ y \& z \to w & (\lambda yz.\, h) & \forall yz\, .h(y,z) = y/z \end{array}$$

If the goal is to construct a program for computing $tg(x)$, then these axioms can be used for verifying that the program

$$\lambda x.\, h(f(x), g(x)),$$

obtained by means of the SSP, performs this task correctly. The complete specification of the result of the synthesis includes the derived formula (goal) and the annotation:

$$x \to w\, (\lambda x.\, h(f(x),g(x)))\ \forall x.h(f(x),g(x)) = tg(x)$$

The situation is different in the case of nested implications. For instance, the axiom

$$(a \to b)\ \&\ (c \to d) \to (x \to y)$$

cannot be in general annotated in the first-order language, because the annotation may contain universally quantified functional variables, which denote the realizations of the subtasks $a \to b$ and $c \to d$. A general form of the annotation for the axiom with subtasks

$$\underline{(\underline{U} \to V)} \to (\underline{X} \to Y)$$

will be the following:

$$\forall x \varphi\, .\, \underline{S}(\varphi)\ \& \underline{P}(x) \to \exists y\, .\, R(\varphi, \underline{x}, y)\, ,$$

where $\underline{S}(\varphi)$ expresses preconditions for the subtasks $\overline{(\underline{U} \to V)}$.

It is still possible to restrict the prof-search to a first-order language after initializing the functional variables to the terms synthesized for them. This is shown in the following example taken from [8].

The methods in the classes for set manipulation defined in Section 2 are annotated as follows:

```
set
    val & sel -> elem (get)
```
$\forall w.\ elem(w) \leftrightarrow \exists z v.\ sel(z)\ \&\ val(v)\ \&\ get(v,z) = w$

$\forall w\ z,v.\ val(v)\ \&\ val(w)\ sel(z) \rightarrow get(v,z) = get(w,z)$

NB! The second formula of the annotation for the class *set* states that we do not distinguish the sets when selecting their elements. This is in accordance with the ideas of SSP about indistinguishability of computed values.

```
subset
    (of.sel -> cond) -> (of.val -> is.val) (F(f))
```
$\forall w.\ is.elem(w) \leftrightarrow$

$\quad \exists z v.\ of.sel(z)\ \&\ of.val(v)\ \&\ get(v,z) = w\ \&\ apply(f,z)$

```
exists
    (in.sel -> cond) -> (in.val -> res) (G(f))
```
$\forall v.\ G(f)(v) \leftrightarrow in.val(v)\ \&\ \exists z.\ in.sel(z)\ \&\ apply(f,z)$

```
intersection
    A.val & B.val -> res.val (spec)
```
$\forall x.\ res.elem(x) \leftrightarrow A.elem(x)\ \&\ B.elem(x)$

The free functional variable `f` appearing in the annotations will be in every particular case substituted by a term synthesized for a subtask. The method of the class `intersection` must be sythesized. In order to verify its correctness, one has to prove its annotation

$$\forall x.\ res.elem(x) \leftrightarrow A.elem(x)\ \&\ B.elem(x).$$

This has been done by means of a resolution method in [8].

5. SYNTHESIS OF ITERATIVE PROGRAMS

A potentially infinite sequence of objects of a class `cc` is specified in a version of NUT as

```
1.. : cc;
```

Let `x` be a component of the class. Then, in the context, where this sequence is specified, the names `#1.x, #2.x,...` denote the `x` component of the first, of the second etc. element of the sequence. Besides that, the names `#curr.x` and `#next.x` denote components of two neighbouring elements of the sequence. These are relative names applicable over the whole sequence.

In order to synthesize an iterative loop on a sequence of objects of a class `cc` for solving the goal `#i.u,...,#i.v -> #j.x,...,#j.y`, where `i`, `j` are fixed numbers, `i < j` we have to find a set `s,...,t` of components of `cc` such that

- all its elements are computable from `u,...,v`
- it contains `x,...,y`
- the following goal is solvable:

```
  #curr.s,...,#curr.t -> #next.s,...,#next.t
```

If this set is found, then the computations are straightforward: first, computing all elements of the set `s,...,t` and thereafter, performing the loop

```
for n:=i step 1 to j-1 do
        #curr.s,...,#curr.t -> #next.s,...,#next.t
od
```

In the case of `j < i`, the roles of `#curr` and `#next` are exchanged. The goal to be solved in the loop is

```
    #next.s,...,#next.t -> #curr.s,...,#curr.t
```

and the synthesized program is

```
for n:=j-1 step -1 to 1 do
    #next.s,...,#next.t -> #curr.s,...,#curr.t
od
```

6. INCREMENTAL PROGRAM DEVELOPMENT BY THE PARTIAL DEDUCTION

Matskin, Komorowski, and Krogstie [9] have formulated the principle of partial deduction for SSP and proved its correctness and completeness. Partial deduction, known in logic programming also as partial evaluation, is applicable for incremental synthesis of programs from specifications. Partial deduction is a derivation of a subgoal from partial input assumptions.

Example:

Let

$$A \rightarrow B\ (\lambda a.\ f)$$
$$A \rightarrow C\ (\lambda\ a.\ k)$$
$$B\ \&\ C \rightarrow D\ (\lambda bc.\ g)$$
$$D \rightarrow E\ (\lambda d.\ h)$$

be axioms of SSP with realizations *f*, *k*, *g*, *h*. Some possible partial deductions for this set of axioms give the following new formulae:

```
A -> D (λa. g(f,k))
B & C-> E (λbc. h(g))
```

These formulae together with their realizations are derived by means of SSP and can be used as additional axioms. This will simplify the program synthesis when the results of partial deduction can be used as parts of the required program, e.g., for the goal $A \rightarrow D$.

ACKNOWLEDGEMENTS

This paper is a survey of results obtained by people who have had interest in the structural synthesis of programs. I wish to thank all of them for this interest. The logic of SSP became clear only due to interest of Grigori Mints to the subject of program construction. His influence on the program synthesis and on the people involved in the present work is much deeper than it can be seen from the references presented here. I am especially indebted to those who have made efforts of experimental testing of ideas, which may seem quite simple in the paper. I wish to thank the authors and supporters of the NUT system, first of all Tarmo Uustalu, Mait Harf, Mikhail Matskin, and Jaan Penjam.

REFERENCES

1. Tyugu, E. Three new-generation software environments. *Comm. ACM*, 1991, **34**, 6, 46–59.
2. Uustalu, T. Extensions of structural synthesis of programs. In *Proc. 6th Nordic Workshop on Programming Theory. BRICS Notes Series NS-94-6* (Engberg, U. et al., eds.). Univ. of Aarhus, 1994, 416–428.
3. Mints, G. Propositional logic programming. In *Towards an Automated Logic of Human Thought* (Hayes, J. E. et al., eds.). *Machine Intelligence*, **12**. Clarendon Press, Oxford, 1991, 17–37.
4. Uustalu, T. and Tyugu, E. Higher-order functional constraint networks. In *Constraint Programming* (Mayoh, E. et al., eds.). *NATO ASI Series F* **131**. Springer, Berlin, 1994, 116–139.
5. Uustalu, T. *Aspects of Structural Synthesis of Programs. TRITA-IT 95:09.* Dept. of Teleinformatics, KTH, 1995.
6. Vene, V., Uustalu, T., and Tyugu, E. Logical semantics of NUT extended with production rules. In *Proc. 5th Symposium on Programming Languages and Software Tools.* Dept. of Computer Science, Univ. of Helsinki, 1997, 145–154.
7. Vickers, S. Geometric logic in computer science. *Proc. 1st Imperial College Dept. of Computing Workshop on Theory of and Formal Methods* (Burn, G. L., Gay, S. J., and Ryan, M. D., eds.). Chelwood Gate, Sussex, UK, 1993, 37–54.
8. Tammet, T. *First Order Correctness Proofs for Propositional Logic Programming. TR CS14/90.* Institute of Cybernetics, Tallinn, 1990.
9. Matskin, M., Komorowski, J., and Krogstie, J. Partial deduction in the framework of structural synthesis of programs. In *Logic of Program Synthesis and Transformation* (Gallagher, J., ed.). *LNCS 1207*. Springer, Berlin, 1997, 239–254.

FUNKTSIONAALPROGRAMMEERIMISE JA PROGRAMMIDE SÜNTEESI PIIRIMAIL

Enn TÕUGU

Kompositsioonilisuse tähtsus programmide konstrueerimisel on leidnud üldist aktsepteerimist. Seoses sellega on relatsioonprogrammeerimisel selged eelised funktsionaalprogrammeerimise ees. Kahjuks puudub relatsioonprogrammeerimises üldine tehnika, mis oleks piisavalt efektiivne olemasolevate funktsionaalprogrammeerimise tehnikatega konkureerimiseks. Artiklis on käsitletud programmide struktuurset sünteesi, s.o. funktsionaalprogrammide sünteesi meetodit, mis on esitatav kõrgemat järku funktsionaalsete kitsenduste, lihtsate tüüpide või intuitsionistliku loogika terminites. Seda meetodit on kasutatud deklaratiivsete keelte realiseerimisel. Nimetatud keeled lubavad spetsifitseerida kontsepte relatsioonidena ja kasutada neid spetsifikatsioonides paindlikumalt kui funktsioone.