Information Processing Letters 116 (2016) 701–705

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Program synthesis by model finding

Alexandre Mota [*], Juliano Iyoda, Heitor Maranhão

Centro de Informática, Universidade Federal de Pernambuco, Av. Jornalista Anibal Fernandes, s/n, Cidade Universitária, CEP 50.740-560, Recife, PE, Brazil

ARTICLE INFO

Article history:
Received 22 September 2015
Received in revised form 7 June 2016
Accepted 9 June 2016
Available online 15 June 2016
Communicated by J.L. Fiadeiro

Keywords:
Program derivation
Program synthesis
Alloy*
Model finding

ABSTRACT

Program synthesis aims to automate the task of programming. In this paper, we present a clear and elegant formulation of program synthesis as an Alloy* specification by applying its model finder to search for a program that satisfies a contract in terms of pre and post-conditions. Our proposal embeds in Alloy* both the syntax and the denotational semantics of Winskel's IMP(erative) language. We illustrate our approach by synthesising Euclid's greatest common divisor algorithm. Our experiments show that our synthesis time is competitive. In addition, Alloy* provides us a great platform for the development of a synthesiser: an elegant synthesiser based on the denotational semantics of a language that can be implemented very quickly.

1. Introduction

Programming is a human activity that is sometimes viewed as Art, sometimes as Math. Program synthesis views programming as Math and aims to automatically produce a program from a specification.

The full automation of the programming activity from a specification is a grand challenge. The automatic translation of a specification to a program can be done in infinitely many ways. The search space is huge. At this point the literature follows different routes:

- The development of synthesisers that are specialised in a particular domain (database [1], reactive systems [2], etc);
- The development of synthesisers that translate a particular language into a SAT/SMT language [3,4];
- And the development of synthesisers that take as input a general purpose high-level specification [5].

In this paper, we follow the approach of a general purpose high-level specification [5]. We use Alloy* [5] and its constraint solver (the Alloy* Analyzer) to create a synthesiser for Winskel's language IMP [6]. Alloy* extends Alloy [7], which is a first-order model finder, to handle quantifications over higher-order structures. The Alloy* higher-order facility is built based on the Counter Example-Guided Inductive Synthesis (CEGIS) [8], which is an approach for solving higher-order synthesis problems.

The main contributions of this paper are:

- The embedding of the syntax and the denotational semantics of a subset[1] of the IMP language in Alloy*;
- A single program synthesiser supporting both program sketches (programs with holes) and a fully automatic synthesis where no sketch is provided;

[*] Corresponding author.
E-mail addresses: acm@cin.ufpe.br (A. Mota), jmi@cin.ufpe.br (J. Iyoda), hpm2@cin.ufpe.br (H. Maranhão).

[1] This subset is representative because it already considers synthesising expressions and program constructors such as loops.

http://dx.doi.org/10.1016/j.ipl.2016.06.003
0020-0190/© 2016 Elsevier B.V. All rights reserved.

- Program synthesis involving the notions of state, conditional commands, sequential compositions and loops following a denotational semantics;
- A case study illustrating the synthesis of Euclid's algorithm for computing the greatest common divisor;
- Experiments illustrating the performance of the synthesiser.

This paper is organised as follows. In Section 2 we show our Alloy* specification of IMP (syntax, well-formedness rules, and semantics) as well as the program synthesis problem. Section 3 illustrates our proposal by synthesising the classical algorithm of Euclid to compute the greatest common divisor. Section 4 discusses related works and Section 5 concludes.

2. Alloy* specification for program synthesis

In what follows we present an embedding of the IMP language, its well-formedness rules, and its semantics in Alloy*.

2.1. The syntax of IMP in Alloy*

Winskel [6] defined **Loc** as a given set of variables (locations). In Alloy*, an abstract signature **sig** defines such given set.

```
abstract sig Loc {}
```

A command Cmd is defined as an abstract syntactic class.

```
abstract sig Cmd {}
```

All concrete commands are defined as subclasses of Cmd. First we define **skip**, which is the command that does nothing.

```
lone sig Skip extends Cmd { }
```

The **extends** keyword, referencing Cmd, is used to establish an inheritance relationship between Cmd and Skip. In this case, **extends** models the grammatical dependency between the non-terminal Cmd and the terminal Skip. The term **lone** constrains the signature to have at most one instance.

An arithmetic expression AExp is either an integer constant (IntVal), integer variable (IntVar), addition (Add), subtraction (Sub) or multiplication (Mult).

```
abstract sig AExp { }
sig IntVal extends AExp { val: one Int }
sig IntVar extends AExp { name: one Loc }
sig Add extends AExp { op1: one AExp, op2: one AExp }...
```

Each class of expression has fields. For instance, an IntVar expression contains the field name that belongs to Loc, and the fields op1 and op2 of an addition are its operands. Subtraction and multiplication are defined similarly to addition.

An assignment $X := a$ is a command whose left-hand side is an integer variable and the right-hand side is an arithmetic expression.

```
sig Assign extends Cmd { lhs: one IntVar, rhs: one AExp }
  { (IntVar<:rhs) ≠ lhs and rhs ∉ IntVal and lhs.name
    ∉ XLoc }
```

The right-hand side must be different from the variable on the left-hand side, must not be a constant, and must not be a read-only variable (XLoc extends Loc and denotes read-only variables).

The sequential composition of commands $C_0; C_1$ becomes the entity SComp. The command C_0 is named as curr (for the current command) and C_1 as next.

```
sig SComp extends Cmd { curr, next: one Cmd }
```

A boolean expression BExp is modelled as an expression of the form lhs OP rhs, where lhs and rhs are arithmetic expressions that are different from auxiliary variables (ALoc)[2]. An OP $\in$ {EQ, NEQ, LEQ, GEQ, GTH}. The operators EQ, NEQ, LEQ, GEQ and GTH denote $=$, $\neq$, $\leq$, $\geq$ and $>$, respectively.

```
abstract sig BExp { lhs, rhs: one AExp }
                  { lhs.name ∉ ALoc and rhs.name ∉ ALoc }
sig EQ extends BExp { }...
```

The operators NEQ, LEQ, GEQ and GTH are defined in a similar way to EQ.

A conditional statement **if** b **then** C_0 **else** C_1 becomes the entity CondS, where we had to change **else** to elsen because the former is a reserved Alloy* keyword.

```
sig CondS extends Cmd { cond: one BExp, then,
                        elsen: one Cmd }
                      { then ≠ elsen }
```

We restrict then to be different from elsen to prevent the synthesis of commands of the form **if** b **then** C **else** C.

Our last and most difficult statement is the while statement. In addition to the usual boolean condition (cond) and body (wbody), we introduce an auxiliary structure (unfold) to unfold the body over the iterations, subject to an invariant.

```
sig While extends Cmd {
    cond: one BExp, wbody: one (Cmd − While),
    unfold: set Expansion, inv: one BExp }
    { (#unfold ≥ 2 ⇒ (∀ disj e1,e2: unfold •
        e1.exp.first.curr.bind ≠ e2.exp.first.curr.bind))
      and cond ≠ inv }
sig Expansion { exp: seq StChg }
    { #exp = #exp.elems and
      (∀i: exp.inds • i ≠ exp.lastIdx
                ⇒ exp[i].next = exp[add[i,1]].curr) }
sig StChg { curr, next: one State }
```

The body wbody is any command except another while (we do not handle nested whiles yet). An Expansion is a sequence of pairs $[(s_0, s_1), (s_1, s_2), ..., (s_{n-1}, s_n)]$, where s_i is a state. The size of an expansion exp is the length of its sequence of state changes exp.elems. Each pair (s_i, s_{i+1}) denotes a state changing (StChg) from s_i to s_{i+1} by running the body of the

[2] An ALoc extends a Loc and denotes auxiliary variables that are forbidden to be used in conditions.