CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories

Fadil Kallat Tristan Schäfer Anna Vasileva
Technical University of Dortmund,
Dortmund, Germany
{fadil.kallat, tristan.schaefer, anna.vasileva}@tu-dortmund.de

We introduce an approach that aims to combine the usage of satisfiability modulo theories (SMT) solvers with the Combinatory Logic Synthesizer (CL)S framework. (CL)S is a tool for the automatic composition of software components from a user-specified repository. The framework yields a tree grammar that contains all composed terms that comply with a target type. Type specifications for (CL)S are based on combinatory logic with intersection types. Our approach translates the tree grammar into SMT functions, which allows the consideration of additional domain-specific constraints. We demonstrate the usefulness of our approach in several experiments.

1 Introduction

In component-based software synthesis, programs are not build from scratch but composed from a repository of typed combinators. Combinators help to reduce the search space so that the inherent complexity of software synthesis problems can be handled. Moreover, additional domain-specific knowledge is contained in the semantic type layer of a repository. The underlying type system is well suited to express feature vectors of programs and software components. A user-specified repository Γ includes typed combinators that represent software components $(c : \sigma)$ where c is the component name and σ is an intersection type [9, 8].

The Combinatory Logic Synthesizer (CL)S is a synthesis framework based on a type inhabitation algorithm for combinatory logic with intersection types [26, 9]. The algorithm searches for terms that are formed from the combinators and have a given target type τ. (CL)S is intended to be used for the automatic composition of software [5, 6, 9, 15, 21]. Besides the synthesis from software components, the (CL)S framework allows the synthesis of data structures, for instance of BPMN 2.0 processes [9] or planning processes [33].

Obviously, the expression of domain-specific knowledge is limited by the underlying type system. Intersection types do not explicitly take the logical connectives conjunction, disjunction and negation into consideration. Moreover, the input-output behaviour of the resulting program cannot be expressed by types. The combinatory approach allows to specify local typing information of a combinator but lacks expressivity regarding the global structure of result terms. For instance, it is not possible to state that a combinator c_0 must contain combinator c_1 anywhere in the subtree of its arguments. In some situations, not all well-formed terms might be considered to be reasonable results. Different terms might also show identical execution results and runtime behaviour.

Software synthesis is an established research topic that offers a broad range of specification formalisms such as examples [16, 17, 28, 32], types [16, 20, 25] or first-order-logic [27, 31]. For this paper, we followed the intuition that the joint usage of (complementary) formalisms can yield a synthesis approach that combines the respective strengths of the underlying techniques. Precisely, we identified SMT to be well working with combinatory logic. There are different possible scenarios to incorporate these techniques. For example, SMT could generate parts of combinators or parametrize synthesized programs. In this paper, we show how to use SMT to filter a complete enumeration of inhabitants. We implemented our approach in a tool called CLS-SMT.

The combinatory logic synthesis yields a tree grammar that describes the set of valid inhabitants. We use this grammar to automatically construct a set of adequate SMT formulas. By solving these formulas, we receive a tree model that represents a word of the grammar. The (possibly infinite) set of inhabitants is further narrowed by introducing domain-specific structural constraints on terms. That way, we can regulate the selection of result programs while avoiding trivial solutions.

The paper is organized as follows: In Section 2 we briefly introduce the composition synthesis framework (CL)S, its underlying theoretical background and the formalism of tree grammars. Section 3 includes a presentation of CLS-SMT and the details about the translation of tree grammars into SMT formulas. In Section 4 we evaluate our approach considering an example for sort programs and a labyrinth example. Section 5 includes an overview of related work. Finally, the conclusion gives a brief summary.

2 Combinatory Logic Synthesizer (CL)S

The developing tool Combinatory Logic Synthesizer (CL)S provides an implementation of a type inhabitation algorithm for combinatory logic with intersection types that is fully integrated into the Scala programming language. The framework is publicly available [7].

The automatic software synthesis is performed by answering the type inhabitation question: $\Gamma \vdash ? : \tau$. The problem of inhabitation asks for all well-typed applicative terms that can be formed from typed combinators in a user-specified set Γ and have a given type τ. Applicative terms are defined as:

$$M,N \ ::= c \mid (MN)$$

A term is constructed by using named component or combinator c and application of M to N, (MN). If there exists a combinatory expression M such that $\Gamma \vdash M : \tau$ then M is called inhabitant of τ. The type expressions that represent the specifications of term M are denoted σ, τ and are defined as follows:

$$\sigma,\tau ::= a \mid \alpha \mid \sigma \to \tau \mid \sigma \ \cap \ \tau$$

Type constants (a) can be native or semantic types. Type variables (α) are substituted with type constants and facilitate generic components. Furthermore, types can be constructed from function types ($\sigma \to \tau$) or intersections ($\sigma \cap \tau$).

There are four rules that control the type inhabitation process. According to these rules, types are assigned to combinatory terms [14]. The first rule (var) allows the usage of any combinator c from the typed repository Γ that has type τ using substitutions. It is defined as follows:

$$\frac{}{\Gamma, c : \tau \vdash c : \mathcal{S}(\tau)} \ (\mathsf{var})$$

Furthermore, it allows to assume that this combinator c has type $S(\tau)$, where S is a well-formed substitution on $\Gamma(c)$ mapping type variables to simple types. The inhabitation problem in general is undecidable. A restriction on variable substitution is needed to ensure decidability [14].

The following rule, arrow elimination ($\to \mathsf{E}$), allows the application of combinators with function types to appropriately typed arguments to form terms.

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau} \; (\to \mathsf{E})$$

The intersection introduction rule ($\cap\mathsf{I}$), shown below, allows to type a term M with two types, if there are proofs that M has type σ and type τ.

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} \; (\cap\mathsf{I})$$

The fourth rule ($\leq$) deals with subtyping.

$$\frac{\Gamma \vdash M : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash M : \tau} \; (\leq)$$

The subtyping rules are based on the Barendregt-Coppo-Dezani-Ciancaglini (BCD) [3] subtyping relation. These include for example:

$$\frac{A_2 \leq A_1 \qquad B_1 \leq B_2}{A_1 \to B_1 \leq A_2 \to B_2}$$

to allow co- and contra-variant subtyping of functions and

$$\frac{}{A \cap B \leq A} \qquad\qquad \frac{}{A \cap B \leq B}$$

to have intersection as the least upper bound. The BCD system is also extended with type constructors, which was proposed in [23, 10].

2.1 Tree Grammar

The (CL)S framework recursively computes all possible solutions in form of tree grammars [11]. We consider the generalized case of *normalized regular tree grammars*, which are well-known from literature [13].

Definition 1 *(Tree Grammars, Tree Grammar Languages)*
A tree grammar G is a 4-tuple $(S, \mathcal{N}, \mathcal{F}, R)$ with

- *a start symbol $S \in \mathcal{N}$*
- *a set $\mathcal{N}$ of nonterminals,*
- *a set $\mathcal{F}$ of terminal symbols,*
- *a set R of productions rules of form $\alpha_1 \mapsto \{c_1(\beta_1, \beta_2, \dots \beta_n),\ c_2(\gamma_1, \gamma_2, \dots \gamma_m)\}$, where $n, m \geq 0$, $\alpha_1, \beta_1, \beta_2, \dots, \beta_n, \gamma_1, \gamma_2, \dots, \gamma_m \in \mathcal{N}$ are nonterminal and $c_1, c_2 \in \mathcal{F}$ are terminal symbols.*

  We consider tree grammars without restriction on the arity of the terminal symbols, e.g. we can have $\alpha_1 \mapsto c_1(\beta_1, \beta_2)$ and $\alpha_2 \mapsto c_1(\beta_1)$ with $\alpha_2 \in \mathcal{N}$.

For a given tree grammar $G = (S, \mathcal{N}, \mathcal{F}, R)$ and nonterminal $\alpha \in \mathcal{N}$, $\mathcal{L}_\alpha(G)$ is the least set closed under the rule

$$\textit{if } \alpha \mapsto c(\beta_1, \beta_2, \ldots, \beta_n) \in R \textit{ and for all } 1 \le k \le n : t_k \in \mathcal{L}_{\beta_k}(G) \textit{ then}$$
$$c(t_1, t_2, \ldots, t_n) \in \mathcal{L}_\alpha(G)$$

We define $\mathcal{L}(G) = \mathcal{L}_S(G)$ to be the language of grammar G.

For request $\Gamma \vdash ? : \tau$, (CL)S constructs a tree grammar $G = (\tau, \mathcal{N}, \mathcal{F}, R)$ where $\tau \in \mathcal{N}$. The right hand sides of rules start with a combinator symbol c where $c \in \mathcal{F}$ is followed by the types of arguments required to obtain the type on the left hand side of the rule by applying the combinator. When (CL)S constructs a tree grammar, we have a word $M \in \mathcal{L}_\tau(G)$. The computed grammar G is *sound* because the word M is well-typed term. Furthermore, G is *complete* because all requested well-typed terms are words of the grammar derived for the target type τ.

2.2 Scala Implementation

The integration of the (CL)S algorithm into Scala allows simple specification of combinators [11]. A typical type specification of the repository Γ for two combinators describing a *start* position and an *up* movement in a game is

$$\Gamma = \{\mathit{start} : \mathit{Pos}(3,4),$$
$$\mathit{up} : (\mathit{Pos}(3,4) \to \mathit{Pos}(3,3)) \cap (\mathit{Pos}(3,3) \to \mathit{Pos}(3,2))\}.$$

Here, arrows are function types and the binary intersection type operator $\cap$ means that a combinator has two types simultaneously. Similar to dependent types [12], specifications can include arbitrary constants and types can encode precomputed function tables. This specification mechanism is Turing complete in general [14], but in practice we use some restrictions, rendering the existence of terms for the type inhabitation problem decidable. In the current version, (CL)S accepts specifications in almost mathematical notation, allowing to state the example for Γ above as:

```
val Gamma = Map("start" -> 'Pos('3, '4),
                "up" -> ('Pos('3, '4) =>: 'Pos('3, '3)) :&:
                        ('Pos('3, '3) =>: 'Pos('3, '2)))
```

It can also extract type information from combinators with implementations attached to them, allowing to enter the combinator *up* from Γ according to the Scala representation in Listing 1. We obtain the specification with native and semantic types: $(\mathit{Pos}(3,4) \to \mathit{Pos}(3,3)) \cap (\mathit{Pos}(3,3) \to \mathit{Pos}(3,2)) \cap (\mathit{Player} \to \mathit{Player})$.

```
@combinator object up {
  def apply(player: Player): Player = player.goUp()
  val semanticType =
    ('Pos('3, '4) =>: 'Pos('3, '3)) :&:
    ('Pos('3, '3) =>: 'Pos('3, '2)))        }
```

Listing 1: Scala representation of a combinator with native and semantic types

The intersection type operator is represented by :&: and the function types by =>:. The signature of apply is automatically translated from its native Scala type. Additional semantic type information is taken as-is and used only to impose more conditions on the use of *up*, which are user specified. The term returned for question $\Gamma \vdash ? : Pos(3,3)$ is $up(start)$, which, when providing combinator implementations, is automatically translated to the method calls `up.apply(start.apply)`. The following tree grammar is the result of the inhabitation:

$$\begin{aligned} G = \{Pos(3,4) &\mapsto \{start()\}, \\ Pos(3,3) &\mapsto \{up(Pos(3,4))\}, \\ Pos(3,2) &\mapsto \{up(Pos(3,3))\}\ \} \end{aligned}$$

3 CLS-SMT

This section describes the key aspects of CLS-SMT. The production rules in the grammar are used to formulate SMT constraints by using uninterpreted functions. Any SMT model satisfying the given constraints represents a tree, which is necessarily a word of the tree grammar.

We define a data structure that represents applicative terms and show how a (CL)S tree grammar can be translated to an adequate SMT formulation.

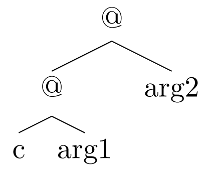
Definition 2 *(Inhabitant Tree)*

An inhabitant tree is a binary tree over integers. Let n denote the finite number of combinators used in the tree grammar and $C \subset \mathbb{N}$ range over $\{1,...,n\}$. With $c \in C$, an inhabitant tree is defined as follows:

$$inhabTree \ = \ 0\ (leftChild\ inhabTree)\ (rightChild\ inhabTree) \mid c$$

Accordingly, the tree's alphabet of vertex labels Σ_V is $\{0\} \cup C$. A vertex labeled 0 is called application node and denoted by @. An @ node has exactly two children (i.e. 0 is a binary symbol), the function is the left child and argument is the right child. All elements of C are constants so that @ nodes are the only elements of the tree that are allowed to have children. A combinator with n arguments is represented by a tree that consists of (at least[1]) n application nodes and the combinator symbol on the leftmost leaf. The n-th argument of a combinator is the right child of the combinators n-th parent. As an example, we consider the term $((c\ (arg1))\ arg2)$, which represents the application of the binary combinator c to the arguments $arg1$ and $arg2$.

We assume that c is encoded as 1, $arg1$ as 2 and $arg2$ as 3. The corresponding inhabitant tree is 0 $(leftChild\ (0\ (leftChild\ 1)\ (rightChild\ 2))\ (rightChild\ 3)$. A visual representation is as follows:



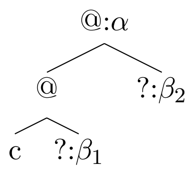
3.1 Constraint Representation

Due to the completeness of the inhabitation algorithm, there is at least one applicative term that can be build from a non-empty tree grammar. Thus, an SMT encoding of the tree grammar on its

[1]more @ nodes could be contained in the subtrees representing the arguments

own will always be satisfiable. There is no need for an encoding of the subtyping relation because subtyping is considered in the inhabitation algorithm. Accordingly, the tree grammar only contains nonterminals representing types and there is a production rule for every nonterminal used.

Let V be the finite set of vertices. The labelling function $inhabitant : V \mapsto \Sigma_V$ can be used for a total representation of a tree if the rules given in Definition 2 are respected. We use the production rules in the tree grammar to formulate structural constraints on the tree. Let $n \in N$ and N denote the set of nonterminals of the grammar. We introduce the partial function $ty : V \mapsto N$, which maps vertices of a tree to a nonterminal representing a type. The information provided by a production rule of the tree grammar can now be used to systematically build constraints for the corresponding subtree. We consider the production rule $\{\alpha \mapsto \{(\mathrm{c}(\beta_1,\ \beta_2)\}\}$ and its incomplete tree representation that is supplemented with the associated nonterminals:



Its possible to derive the following constraints from this production rule. Let i denote the root node of the applicative composition of the combinator and its arguments. If node i has type represented by nonterminal α then the vertex `(leftChild (leftChild i))` must be c, the first argument (at position `(rightChild (leftChild i))`) must be typed according to β_1 and the second argument (at `(rightChild i)`) must be typed corresponding to β_2. The constraints for subtrees denoted by β_1 and β_2 can be formulated accordingly. Following this approach, the contents of a tree grammar can be translated into SMT constraints. Adequate assertions are formulated and supplied to the SMT solver to find implementations for the uninterpreted functions *inhabitant* and *ty*. We currently use Z3 from Microsoft Research [24] to solve our formulation with the background theory LIA [4] (i.e. the linear fragment of the theory of Integers). A more detailed look at the translation will be given in the next section.

3.2 Grammar Translation

We translate the grammar by applying TRANSLATE_PRODUCTION_RULE shown in Algorithm 1 to every production rule of the grammar. The algorithm produces SMT boolean expressions that must evaluate to true for all vertices of a valid tree. We make use of the aforementioned functions *inhabitant* and *ty* to formulate these constraints. The set of constraint functions is incorporated in an assertion with a $forall$ expression where the universal quantified variable i represents the vertices. Consequently, every solution found by the SMT solver must be a word of the grammar.

Inside TRANSLATE_PRODUCTION_RULE, the function `Translate_Combinator` is applied to every possible combinator listed in this specific production rule. The resulting set of boolean expressions is joined with the `xor` connective as we must use one combinator subtree exclusively at a given type annotated vertex. For the sake of readability, we assume that `xor` and `and` are applicable to sets.

An n-ary combinator is translated by using the universal quantified variable i and its associated children to describe the vertices of the respective subtree. The labelling is formulated by placing constraints on the *ty* and *inhabitant* functions. We reverse the list of nonterminals *args* that describes the required types of a combinator's arguments in order to address the structure of inhabitant trees. That way, we can start at the root node of the current subtree and build successive address terms for each loop iteration by applying `leftChild` to the current address term. The complete structure of the subtree must satisfy all constraints that were produced in the loop, so we return the corresponding conjunction. After translating the grammar rules, we also include a root node constraint. It states that *ty* must map node 1 of the tree to the nonterminal representing the synthesis goal type.

Algorithm 1 Production Rule Translation

```
function TRANSLATE_PRODUCTION_RULE($typeId, values$)
    $xorSet \leftarrow \varnothing$
    for all ($combinator, parameters$) in values do
        $cTransl \leftarrow$ TRANSLATE_COMBINATOR(combinator, parameters)
        $xorSet \leftarrow xorSet \cup cTransl$
    end for
    return (ite (= (ty i) $typeId$) (xor $xorSet$) true)
end function

function TRANSLATE_COMBINATOR($combinator, args$)
    $constrSet \leftarrow \varnothing$
    $currentAddress \leftarrow i$
    $pList \leftarrow args.reverse$
    for all p in pList do
        $constrSet \leftarrow constrSet \cup$ (= (ty (rightChild $currentAddress$)) p)
        $constrSet \leftarrow constrSet \cup$ (= (inhabitant $currentAddress$) 0)
        $currentAddress \leftarrow$ (leftChild $currentAddress$)
    end for
    $combinatorConstraint \leftarrow$ (= (inhabitant $currentAddress$) combinator)
    $combinedSet \leftarrow combinatorConstraint \cup constrSet$
    return (and ($combinedSet$))
end function
```

Any tree model M^* that satisfies these constraints represents a word M of the grammar and every word M can be translated to a model M^* that satisfies these constraints. The translation is straight-forward and is thus be omitted. Let φ denote the conjunction of the constraints and τ denote the inhabitation target type, then: $M^* \vDash_{LIA} \varphi \Leftrightarrow M \in \mathcal{L}_\tau(G)$.

4 Experiments

In this section, we discuss the advantages and the usefulness of our approach by means of a composition of sort programs and a path finding scenario.

4.1 Sort

We consider a small repository Γ shown in Fig. 1 that can be used to compose sort programs. It contains a sort combinator for lists that applies a function to each element before performing the sorting. The *id* combinator typed $\alpha \to \alpha$ can be used if we want to sort the unmodified list values. Moreover, the inverse function can be applied to double values. Further combinators could include the *abs* function to compare absolute values or a *dist* combinator to calculate the distance to a given value.

$$\begin{aligned}\Gamma = \{\; & values : List(double),\\ & id : \alpha \to \alpha,\\ & inv : double \to double,\\ & sortmap : (\alpha \to \alpha) \to List(\alpha) \to SortedList(\alpha),\\ & min : double \to SortedList(double) \to minimal \cap double,\\ & default : double \;\}\end{aligned}$$

Figure 1: Repository for the sort example

In some cases, it might be required to sort a *double* list and additionally determine its minimal value. The corresponding combinator *min* will be implemented by extracting the first value of a sorted list (assuming that we always sort in an ascending order). The result type of *min* is an intersection of *minimal* and *double*. For empty lists, a default value will be returned. In this example, such a value is held in the component $default$, which has the type *double*. The inhabitation request $\Gamma \vdash ? : minimal \cap double$ yields the following grammar G:

$$\begin{aligned}G = \{ & SortedList(double) \mapsto \{sortmap(double \to double, List(double))\},\\ & minimal \cap double \mapsto \{id(minimal \cap double), min(double, SortedList(double))\},\\ & double \mapsto \{id(double), default(), inv(double), min(double, SortedList(double))\},\\ & double \to double \mapsto \{id(), inv()\}\\ & List(double) \mapsto \{id(List(double)), values()\} \;\}\end{aligned}$$

Figure 2: Tree grammar for the sort example, $\Gamma \vdash ? : minimal \cap double$

A double value can be formed by applying *id* or *inv* to any term with type double. Obviously, terms like *inv* and *id* can be applied an arbitrary number of times to arguments of type double. Thus, the range of terms with type double is infinite. Moreover, a term typed $minimal \cap double$ can also be used as the first argument of the *min* operator. The grammar describes all well-formed solutions that comply to the target type. However, it is clearly not desirable to compose infinite range of trivial solutions. With extensions formulated as SMT constraints, we can further filter the result set without specializing Γ too much.

In order to avoid trivial solutions, we specify *id* and *inv* to be used only as arguments. Moreover, the first argument of *min* must be a terminal. Given the indices 2, 3 and 5 for the combinators *id*, *min* and *inv*, the following assertions are added to the SMT script:

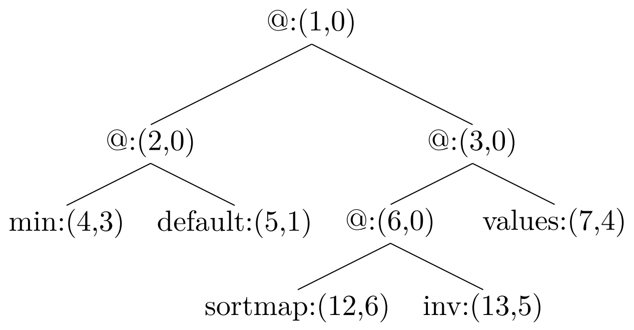
```
(assert (forall ((i Int)) (not (= (inhabitant (leftChild i)) 2))))
(assert (forall ((i Int)) (not (= (inhabitant (leftChild i)) 5))))
(assert (forall ((i Int))
        (ite (= (inhabitant (leftChild i)) 3)
                (not (= (inhabitant (rightChild i)) 0)) true)))
```

With these constraints at hand, only two valid solutions are found for the inhabitation request $\Gamma \vdash ? : minimal \cap double$:

$$((min\ default)\ ((sortmap\ inv)\ values))\ \text{and}$$
$$((min\ default)\ ((sortmap\ id)\ values))$$

The combinator *min* is applied to the terms yielded by the combinators *default* and *sortmap*. For this particular example, the combinator mapping in the table shown below was used. In order to illustrate the first result term as a tree, we use the following labelling pattern: *combinator name* : (*vertex id*, *combinator id*)

name	id
default	1
id	2
min	3
values	4
inv	5
sortmap	6



4.2 Labyrinth Example

In the following labyrinth example, it is possible to go *up*, *down*, *left* or *right*, if the new position is not occupied by obstacles [11]. Fig. 3 illustrates a 3 × 4 labyrinth example. The starting position is $(0, 2)$ (shown as $\bullet$) and the goal position $(1, 0)$ (shown as $\bigstar$).

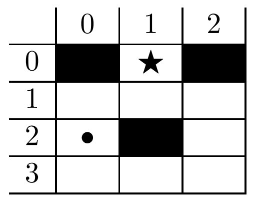


Figure 3: Labyrinth example

The repository with typed combinators for this example is represented in Fig. 4.

$$\begin{aligned}
\Gamma_{Lab} = \{\ left : & (Pos(1,1) \to Pos(0,1)) \cap Pos(2,1) \to Pos(1,1)) \ \cap \\
& (Pos(1,3) \to Pos(0,3)) \cap (Pos(2,3) \to Pos(1,3)), \\
right : & (Pos(0,1) \to Pos(1,1)) \cap (Pos(1,1) \to Pos(2,1)) \ \cap \\
& (Pos(0,3) \to Pos(1,3)) \cap (Pos(1,3) \to Pos(2,3)), \\
up : & (Pos(0,3) \to Pos(0,2)) \cap (Pos(2,3) \to Pos(2,2)) \ \cap \\
& (Pos(1,1) \to Pos(1,0)) \cap (Pos(0,2) \to Pos(0,1)) \ \cap \\
& (Pos(2,2) \to Pos(2,1)), \\
down : & (Pos(1,0) \to Pos(1,1)) \cap (Pos(0,1) \to Pos(0,2)) \ \cap \\
& (Pos(2,1) \to Pos(2,2)) \cap (Pos(0,2) \to Pos(0,3)) \ \cap \\
& (Pos(2,2) \to Pos(2,3)), \\
start : & Pos(0,2)\ \}
\end{aligned}$$

Figure 4: Repository for the labyrinth example shown in Fig. 3

The combinators *up, down, left*, and *right* can be used to go from position $Pos(x,y)$ to an accessible neighbouring position. The types $Pos(x,y)$ represent the column and row positions. For example, combinator *left* can be used to go from position $Pos(1,1)$ to position $Pos(0,1)$ as well as from $Pos(2,1)$ to $Pos(1,1)$, from $Pos(1,3)$ to $Pos(0,3)$, and from $Pos(2,3)$ to $Pos(1,3)$. The combinator *start* provides the starting position.

To get all possible paths from start (0,2) to goal position (1,0), we ask for:

$$\Gamma \vdash ? : Pos(1,0)$$

For this goal position the algorithm computes the grammar shown in Fig. 5.

$$\begin{aligned}
G = \{Pos(1,0) \mapsto\ & \{up(Pos(1,1))\}, \\
Pos(1,1) \mapsto\ & \{right(Pos(0,1)),\ left(Pos(2,1)),\ down(Pos(1,0))\}, \\
Pos(1,1) \mapsto\ & \{up(Pos(0,2)),\ left(Pos(1,1))\}, \\
Pos(2,1) \mapsto\ & \{up(Pos(2,2)),\ right(Pos(1,1))\}, \\
Pos(2,2) \mapsto\ & \{down(Pos(2,1)),\ up(Pos(2,3))\}, \\
Pos(0,1) \mapsto\ & \{up(Pos(0,2)),\ left(Pos(1,1))\}, \\
Pos(0,3) \mapsto\ & \{down(Pos(0,2)),\ left(Pos(1,3))\}, \\
Pos(0,2) \mapsto\ & \{down(Pos(0,1)),\ up(Pos(0,3)),\ start()\}, \\
Pos(1,3) \mapsto\ & \{left(Pos(2,3)),\ right(Pos(0,3))\}, \\
Pos(2,3) \mapsto\ & \{down(Pos(2,2)),\ right(Pos(1,3))\}\}
\end{aligned}$$

Figure 5: Tree grammar for the labyrinth example

For the path going *up*, *right*, and *up* the algorithm constructs a term $up(right(up(start)))$. In this example, there are also terms that represent trivial paths with cycles. For example:

$$up(right(up(down(up(down(up(start))))))),$$
$$down(up(up(right(up(start))))), \ldots$$

By means of SMT solvers, we can restrict the number of solutions computed by (CL)S in order to avoid trivial terms. For example, we can decide, which combinators have to be used and how often. As presented in Section 3 we translate the computed tree grammar (s. Fig. 5) to SMT expressions by means of algorithm 1.

In order to filter the inhabitants, we consider domain-specific constraints. We are able to select, which combinators should be used in the solution. For instance, Fig. 6 shows a formula that states a term should not include combinator *down* (translated as `(= (inhabitant i) 1)`). This way, we constrain the usage of certain combinator. In this particular example (see Fig. 3), we might want to avoid the *down* combinator, because the robot has to get to the top-right goal position.

```
(assert (forall ((i Int)) (not (= (inhabitant i) 1))))
```

Figure 6: Assertion for filtering of combinator

We reduce the number of cycles and define the order of usage of the combinators in order to avoid unnecessary paths. For example, we can formulate a constraint that forbids the application of combinator *down* (index 1) to combinator *up* (index 2) and vice versa. The same applies to combinators *left* (index 3) and *right* (index 4). Fig. 7 shows the definition of this rule.

```
(assert (forall ((i Int))
 (and
  (not (and (= (inhabitant (leftChild i)) 3)
  (= (inhabitant (leftChild (rightChild i))) 4)))
  (not (and (= (inhabitant (leftChild i)) 4)
  (= (inhabitant (leftChild (rightChild i))) 3)))
  (not (and (= (inhabitant (leftChild i)) 2)
  (= (inhabitant (leftChild (rightChild i))) 1)))
  (not (and (= (inhabitant (leftChild i)) 1)
  (= (inhabitant (leftChild (rightChild i))) 2))))
 ))
```

Figure 7: Formula for definition of order

5 Related Work

Type-theoretical specification

There are various approaches to solve synthesis problems by means of type theory. For instance, Polikarpova et al. synthesized recursive functions satisfying a specification in the form of polymorphic refinement types [25]. Zdancewic et al. demonstrated that examples in example-directed synthesis can be interpreted as refinement types [16]. They provided an example-based specification language by using intersection types with singletons. In contrast, (CL)S expresses semantic specifications with intersection types. Kuncak et al. used type inhabitation in the simply typed lambda calculus to support developers by generating a list of valid expressions of a given type for code completion [20].

SMT

In the last decades, there have been many approaches using SMT solvers for synthesis. A common property of those methodologies is the use of syntactic constraints and a correctness specification. In 2006, preliminary work in template-based synthesis was undertaken by Solar-Lezama et al. [29]. In *Sketching*, a partial implementation is given and synthesis completes missing parts by considering a specification of the desired functionality [29]. Following this idea, loop-free bitvector programs [18] and deobfuscating programs [22] were synthesized in a component-based manner. In contrast to our work, desired functionality and components were specified as logical relations between the input and output variables [18, 22]. Another approach in SMT based synthesis is programming by examples. A user specifies the behaviour of the desired program by a number of input-output examples [19]. Singh and Gulwani transformed strings and data types in spreadsheets [17, 28] and Udupa et al. were able to synthesize protocols from a given skeleton and examples [32].

In 2013, a number of researchers picked up the main ideas of the projects above to formulate the problem of syntax-guided synthesis (SyGuS) [1]. The Counterexample-Guided Inductive Synthesis (CEGIS) architecture describes how SyGuS problems can be tackled by learning from counterexamples provided by a verification oracle, which is often implemented by off-the-shelf SMT solvers [1].

Most of the synthesis algorithms based on CEGIS variants are solving $\exists\forall$-formulas iteratively using SMT solvers [1]. Similar to Reynolds et al. we consider synthesis as a theorem-proving problem. In our case, the problem is solved in combinatory logic and later refined by a SMT solver, whereas in [27] the problem is solely solved within the SMT solver. The main difference is the way of specification. Like in many traditional synthesis approaches [16, 17, 28, 32], targets in [27] are specified by using properties of executed programs. More specifically, relations on inputs and outputs are defined. This allows for a fine-granular specification on program behaviour, but it is hard to control the structure of synthesized programs. It can also be hard to specify the program behaviour in the SMT solver, which becomes especially apparent in the presence of side effects or exceptions. In (CL)S, these concerns are hidden behind the interfaces of types. Types are particularly easy to define, because they already exist in most programming languages and do not need to be specified just for synthesis. They can encode taxonomic concepts via semantic types and subtyping, which is usually a very natural way of expression [30]. In future work, it might be interesting to consider bridging the gap between behavioural and type-based specifications. In particular, the approach in [27] could be used to synthesize the implementation for individual combinators, which are then composed by (CL)S and CLS-SMT.

6 Conclusion

In our work we combined Combinatory Logic Synthesis and Satisfiability Modulo Theories in a tool called CLS-SMT. In this way, we are able to compensate limitations of one technology by taking advantage of the other and vice versa. The synthesis framework (CL)S generates a tree grammar from a given repository of typed components that contains domain-specific knowledge. We should emphasize that the tree grammar is *complete* and describes all well-formed solutions.

CLS-SMT translates the grammar into SMT formulas and further domain-specific constraints are added. The SMT solver Z3 finds a model considering the translated tree grammar and constraints.

By having further constraints formulated as SMT formulas, we are able to restrict inhabitants without restricting the types of the (CL)S component repository. That way, we benefit from the expressiveness of first-order logic and background theories. Combinatory logic synthesis reduces the search space of the SMT solver. In general, SMT considers this structure of the programs, whereas components in (CL)S contain domain-specific details. Combinatory logic with intersection types is a Turing complete formalism that allows to define semantic taxonomies based on subtyping [9].

Although SMT solvers are highly efficient through decades of research and improvements, handling quantified formulas is still challenging. Congruence Closure with Free Variables (CCFV) [2] is a framework that is based on the E-ground (dis)unification problem and unifies major instantiation techniques in SMT solving. Experimental evaluation shows that CCFV improved the performance of the solvers CVC4 and veriT significantly, so that the former outranks the state-of-the-art in instantiation based SMT solving. Within our research, the replacement of solvers is possible with reasonable effort due to the SMT-LIB standard. Further performance enhancements could be achieved by exploring the usage of data types to express the tree.

We have applied CLS-SMT to synthesize sort programs and motion plans. Motion planning problems are an interesting topic for program synthesis because of the associated scaling problems. Our examination shows that synthesis of small motion plans is successful. On the other hand, we found that larger examples do not scale properly. Our approach is well-suited for motion plans with up to 10×10 tiles. Scaling problems do not apply to other use cases such as the sort example. Future work considers an investigation of motion planning problems with multiple robot instances and obstacles.

Acknowledgement. The work presented in this paper was partly funded by the GRK 2193 (`www.grk2193.tu-dortmund.de/de/`) and the Center of Excellence for Logistics and IT (`www.leistungszentrum-logistik-it.de/`) located in Dortmund.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak & A. Udupa (2013): *Syntax-guided synthesis.* In: *2013 Formal Methods in Computer-Aided Design*, pp. 1–8, doi:`10.1109/FMCAD.2013.6679385`.

[2] Haniel Barbosa, Pascal Fontaine & Andrew Reynolds (2017): *Congruence Closure with Free Variables.* In Axel Legay & Tiziana Margaria, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 10206, Springer, Berlin, Heidelberg, pp. 214–230, doi:`10.1007/978-3-662-54580-5_13`.

[3] H. P. Barendregt, M. Coppo & M. Dezani-Ciancaglini (1983): *A Filter Lambda Model and the Completeness of Type Assignment.* *Journal of Symbolic Logic* 48(4), pp. 931–940, doi:`10.2307/2273659`.

[4] Barrett, C., Fontaine, P., Tinelli, C.: *SMT-LIB Logics.* Available at `http://smtlib.cs.uiowa.edu/logics.shtml`.

[5] Jan Bessai, Tzu-Chun Chen, Andrej Dudenhefner, Boris Düdder, Ugo de'Liguoro & Jakob Rehof (2018): *Mixin Composition Synthesis based on Intersection Types.* *Logical Methods in Computer Science* Volume 14, Issue 1, doi:`10.23638/LMCS-14(1:18)2018`.

[6] Jan Bessai, Boris Düdder, George T. Heineman & Jakob Rehof (2015): *Combinatory Synthesis of Classes Using Feature Grammars.* In: *Revised selected papers of the 12th International Conference on Formal Aspects of Component Software*, pp. 123–140, doi:`10.1007/978-3-319-28934-2_7`.

[7] Jan Bessai, Boris Düdder, Geroge T. Heineman et al. (2018): *(CL)S Framework.* Available at `http://www.combinators.org`. Accessed: 2018-04-30.

[8] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens & Jakob Rehof (2014): *Combinatory Logic Synthesizer.* In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, *Lecture Notes in Computer Science* 8802, Springer, Berlin, Heidelberg, pp. 26–40, doi:`10.1007/978-3-662-45234-9_3`.

[9] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens & Jakob Rehof (2016): *Combinatory Process Synthesis.* In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pp. 266–281, doi:`10.1007/978-3-319-47166-2_19`.

[10] Jan Bessai, Jakob Rehof & Boris Düdder (2019): *Fast Verified BCD Subtyping.* In: *Models, Mindsets, Meta: The What, the How, and the Why Not?*, *Lecture Notes in Computer Science* 11200, Springer, Cham, [S.l.], pp. 356–371, doi:`10.1007/978-3-030-22348-9_21`.

[11] Jan Bessai & Anna Vasileva (2018): *User Support for the Combinator Logic Synthesizer Framework.* *Electronic Proceedings in Theoretical Computer Science* 284, pp. 16–25, doi:`10.4204/EPTCS.284.2`.

[12] Edwin Brady (2017): *Type-driven development with Idris.* Manning Publications Co, Shelter Island, NY.

[13] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications.* Available online: `http://www.grappa.univ-lille3.fr/tata`. Release October, 12th 2007.

[14] Boris Düdder, Moritz Martens, Jakob Rehof & Paweł Urzyczyn (2012): *Bounded Combinatory Logic.* In Patrick Cégielski & Arnaud Durand, editors: *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, *LIPIcs* 16, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 243–258, doi:`10.4230/LIPIcs.CSL.2012.243`.

[15] Boris Düdder, Jakob Rehof & George T. Heineman (2015): *Synthesizing type-safe compositions in feature oriented software designs using staged composition.* In: *Proceedings of the 19th International Conference on Software Product Lines*, pp. 398–401, doi:`10.1145/2791060.2793677`.

[16] Jonathan Frankle, Peter-Michael Osera, David Walker & Steve Zdancewic (2016): *Example-directed Synthesis: A Type-theoretic Interpretation.* In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, ACM, New York, NY, USA, pp. 802–815, doi:`10.1145/2837614.2837629`.

[17] Sumit Gulwani, William R. Harris & Rishabh Singh (2012): *Spreadsheet data manipulation using examples.* *Communications of the ACM* 55(8), pp. 97–105, doi:`10.1145/2240236.2240260`.

[18] Sumit Gulwani, Susmit Jha, Ashish Tiwari & Ramarathnam Venkatesan (2011): *Synthesis of Loop-free Programs.* In: *Proceedings of PLDI'11*, ACM, p. 62, doi:`10.1145/1993498.1993506`.

[19] Sumit Gulwani, Oleksandr Polozov & Rishabh Singh (2017): *Program Synthesis.* *Foundations and Trends® in Programming Languages* 4(1-2), pp. 1–119, doi:`10.1561/2500000010`.

[20] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj & Ruzica Piskac (2013): *Complete Completion Using Types and Weights.* In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, ACM, New York, NY, USA, pp. 27–38, doi:`10.1145/2491956.2462192`.

[21] George T. Heineman, Jan Bessai, Boris Düdder & Jakob Rehof (2016): *A Long and Winding Road Towards Modular Synthesis.* In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pp. 303–317, doi:10.1007/978-3-319-47166-2_21.

[22] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia & Ashish Tiwari (2010): *Oracle-guided component-based program synthesis.* In Jeff Kramer, Judith Bishop, Prem Devanbu & Sebastian Uchitel, editors: *ACM/IEEE 32nd International Conference on Software Engineering, 2010*, IEEE, NJ, USA, p. 215, doi:10.1145/1806799.1806833.

[23] Olivier Laurent (2018): *Intersection Subtyping with Constructors.* In Michele Pagani & Sandra Alves, editors: *Proceedings DCM 2018 and ITRS 2018*, *EPTCS* 293, Oxford, UK, pp. 73–84, doi:10.4204/EPTCS.293.6.

[24] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver.* In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, *Lecture Notes in Computer Science* 4963, Springer, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[25] Nadia Polikarpova, Ivan Kuraj & Armando Solar-Lezama (2016): *Program Synthesis from Polymorphic Refinement Types.* In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, ACM, New York, NY, USA, pp. 522–538, doi:10.1145/2908080.2908093.

[26] Jakob Rehof (2013): *Towards Combinatory Logic Synthesis.* In: *BEAT 2013, 1st International Workshop on Behavioural Types*, ACM.

[27] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett & Morgan Deters (2017): *Refutation-based synthesis in SMT.* *Formal Methods in System Design*, doi:10.1007/s10703-017-0270-2.

[28] Rishabh Singh & Sumit Gulwani (2016): *Transforming spreadsheet data types using examples.* In Rastislav Bodik & Rupak Majumdar, editors: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, ACM, New York, NY, USA, pp. 343–356, doi:10.1145/2837614.2837668.

[29] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia & Vijay Saraswat (2006): *Combinatorial sketching for finite programs.* In John Paul Shen, editor: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ACM, New York, NY, USA, p. 404, doi:10.1145/1168857.1168907.

[30] Bernhard Steffen, Tiziana Margaria & Michael von der Beeck (1997): *Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach.* In: *ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97).*

[31] Ashish Tiwari, Adrià Gascón & Bruno Dutertre (2015): *Program Synthesis Using Dual Interpretation.* In Amy Felty & Aart Middeldorp, editors: *Automated deduction – CADE-25, LNCS sublibrary. SL 7, Artificial intelligence* 9195, Springer, Cham, pp. 482–497, doi:10.1007/978-3-319-21401-6_33.

[32] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin & Rajeev Alur (2013): *TRANSIT: Specifying Protocols with Concolic Snippets.* *SIGPLAN Not.* 48(6), pp. 287–296, doi:10.1145/2499370.2462174.

[33] Jan Winkels, Julian Graefenstein, Tristan Schäfer, David Scholz, Jakob Rehof & Michael Henke (2018): *Automatic Composition of Rough Solution Possibilities in the Target Planning of Factory Planning Projects by Means of Combinatory Logic.* In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, *Lecture Notes in Computer Science* 11247, Springer, Cham, pp. 487–503, doi:10.1007/978-3-030-03427-6_36.