Combinatory Logic Synthesizer

Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens, and Jakob Rehof

Technical University of Dortmund, Department of Computer Science

Abstract. We present Combinatory Logic Synthesizer (CL)S, a type-based tool to automatically compose larger systems from repositories of components. We overview its underlying theory, combinatory logic with intersection types, and exemplify its application to synthesis. We describe features and architecture of the tool and our plans for its ongoing and future development. Finally, we present some use cases in ongoing work, especially in the context of synthesis for Object Oriented Software.

1 Introduction

Combinatory logic synthesis [1] is a type-theoretic approach towards synthesis from specified components in a repository. Components are represented by typed combinators $(X : \rho)$ where X is the component name and ρ is an intersection type [2] representing both the compontent's actual interface type and semantic information describing the component's intended usage. The question whether there is a composition e of components in the repository such that e satisfies a certain goal specification τ corresponds to the *relativized inhabitation problem* in combinatory logic — given a set Γ of typed combinators and a goal type τ, does there exist an applicative term e (referred to as an *inhabitant*) of type τ under the type assumptions in Γ? Algorithms solving the relativized inhabitation problem for certain combinatory logics have been given [3–5], thus laying the foundation for a tool-realization of combinatory logic synthesis.

In this paper we report on current work on and on experiments with Combinatory Logic Synthesizer, (CL)S,[1] an end-to-end prototypical tool, providing user-support in design of repositories, a core inhabitation routine based on the above mentioned inhabitation algorithms, optimizing heuristics of the inherently complex core algorithms, and means of translating synthesized compositions into executable code in various implementation languages.

The paper is structured as follows. Section 2 develops background information on combinatory logic synthesis, provides an overview of related work and discusses a motivating example. In Section 3 we describe the architecture and the current state of development of (CL)S. In Section 4 we discuss ongoing work and plans for future development. Section 5 concludes by presenting some examples from ongoing work on applications of our tool, particularly within the context of Object Oriented software by means of Java Virtual Machine languages.

[1] `http://www-seal.cs.tu-dortmund.de/seal/pages/research/cls_en.shtml`

2 Background and Related Work

The work presented in this paper is closely related to current movements[2] towards component-based synthesis, where synthesis is considered relative to a given collection (library, repository) of components rather than aiming at construction of a system "from scratch". Possible benefits of relativizing synthesis to component collections include the exploitation of design intelligence and abstraction (in the form of abstract component interfaces) embodied by components. Moreover, modern development scenarios increasingly depend on extended usage of components. However, for component-oriented synthesis to work well, components presumably need to be designed for composition.

Synthesis approaches can be distinguished by the model of computation as well as the methods of specification assumed. One line of work is characterized by the usage of temporal logic and automata theoretic models, whereas another is characterized by the usage of deductive methods in program logics (e.g., [6]) and in type theory. A recent comprehensive introduction and survey on program synthesis is presented in [7] providing a categorization scheme for synthesis methods. Within this scheme combinatory logic synthesis is classifiable as functional synthesis with semantic candidate spaces.

Recently, component-orientation was promoted in the automata theoretic approach by Lustig and Vardi [8]. Combinatory logic synthesis was proposed in [1] as a deductive, type-based approach to synthesis from components in a repository, where repositories are regarded as type environments in combinatory logic [9]. Component interfaces are specified semantically using intersection types [2]. The logical basis for combinatory logic synthesis is the relativized inhabitation problem, in that algorithms for solving this problem can be used to automatically synthesize component compositions (inhabitants).

The pioneering work on synthesizing linear process models (sequential composition of calls to components) in [10] combines temporal constraints with types and subtyping to capture taxonomic hierarchies. Combinatory logic synthesis is related to adaptation synthesis via proof counting discussed in [11, 12], where semantic types are combined with proof search in a specialized proof system. In particular, we follow the approach in [11, 12] in using semantic specifications at the interface level, where semantic specifications are assumed to correctly describe properties of the component (checking that this is indeed so is regarded as an orthogonal issue). The idea of adaptation synthesis in [11] is related to our notion of composition synthesis, however our logic is different, our design of semantic types with intersection types is novel, and the algorithmic methods are different. Semantic intersection types can be compared to refinement types [13], but semantic types do not need to stand in a refinement relation to implementation types. Still, refinement types are a great source of inspiration for how semantic types can be used in specifications in many interesting situations.

[2] `http://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=14232`.

2.1 Composition Synthesis

In its minimal form, composition synthesis consists of a single logical rule:

$$\frac{\Gamma \vdash F : \tau' \to \tau \quad \Gamma \vdash G : \tau'}{\Gamma \vdash (F\ G) : \tau}(\to\text{E})$$

The (→E)-rule (viewed logically, under the Curry-Howard isomorphism, this is modus ponens) forms the simplest logical model of applicative composition of *named component specifications* $(X : \rho) \in \Gamma$ from a *repository* Γ, satisfying *goal* τ. With fixed Γ and τ as part of the input, the *inhabitation problem* is the decision problem

$$\exists e.\ \Gamma \vdash e : \tau?$$

— does there exist a composition e from repository Γ with $\Gamma \vdash e : \tau$? An inhabitation algorithm is used to *construct* or synthesize a composition e from Γ and τ. The inhabitation problem is the foundation for *automatic synthesis* and is inherently *component-oriented*.

In a type-oriented approach to composition synthesis, types (τ) take the role of specifications of named components represented by terms (e):

$$\begin{aligned}\text{Types } \tau, \tau' &::= a \mid \alpha \mid \tau \to \tau' \\ \text{Terms } e, e' &::= X \mid (e\ e')\end{aligned}$$

Types are constructed from constants (a), variables (α), or function types ($\tau \to \tau'$). Terms are constructed by using named components or combinators X and using application of e to e', $(e\ e')$. An additional rule (var) is added to allow schematic instantiations of combinator types (component specifications) using substitutions.

$$\frac{\text{Substitution } S}{\Gamma, (X : \tau) \vdash X : S(\tau)}(\text{var}) \qquad \frac{\Gamma \vdash e : \tau' \to \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau}(\to\text{E})$$

From a logical point of view, this system is a Hilbert-style presentation of a minimal propositional logic based on implicit schematism (var) and modus ponens (→E).

2.2 Example for Combinatory Logic Synthesis

The idea of combinatory logic synthesis is demonstrated by the following example presenting a scenario for tracking containers in logistics (see also [1] where this example was introduced). Figure 1 presents a repository Γ containing existing (API-)functions for tracking containers, where `R` is the data-type real. Function `0` returns a tracking object `TrObj`. Given a tracking object `TrObj`, function `Tr` returns a triple of which the first entry is a coordinate of the tracking object, followed by time information, and the current temperature of the tracking object. The function `pos` projects the position and time from such a triple. Function `cdn` projects the coordinate and functions `fst` and `snd` project the first resp. second

```
O      : TrObj
Tr     : TrObj → D((R, R), R, R)
pos    : D((R, R), R, R) → ((R, R), R)
cdn    : ((R, R), R) → (R, R)
fst    : (R, R) → R
snd    : (R, R) → R
tmp    : D((R, R), R, R) → R
cc2pl  : ((R, R), R) → ((R, R), R)
cl2fh  : R → R
```

Fig. 1: Repository Γ containing functions for tracking containers in logistics

entry in a pair like a coordinate. Function `tmp` returns the temperature. Two additional conversion functions `cc2pl` and `cl2fh` are contained, that convert Cartesian to polar coordinates and temperature from Celsius to Fahrenheit.

Figure 2 presents a taxonomy that describes semantic concepts, noted in blue, in our scenario. Dashed lines denote a *has-a* relationship whereas continuous lines denote an *is-a* relationship. In particular, *Trackdata* contains a position *Pos* and a temperature *Temp* that can be measured in Celsius *Cel* or Fahrenheit *Fh*.

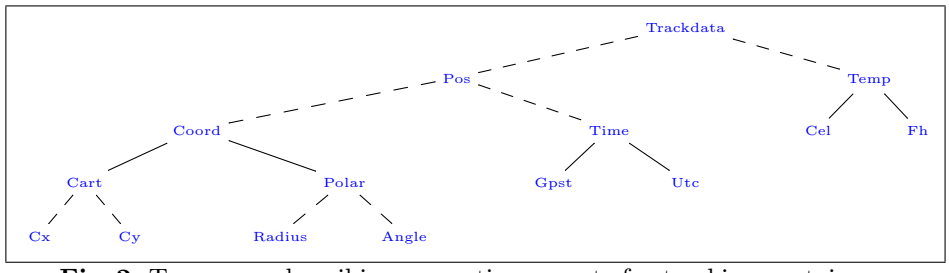

Fig. 2: Taxonomy describing semantic concepts for tracking containers.

None of the semantic information is explicitly included in Γ but only in its verbose description. A central idea in our approach to combinatory logic synthesis is that intersection types, introduced in [2], can be used to semantically refine the specification of functions contained in the repository Γ, for example by adding abstract conceptual information from a taxonomy. Figure 3 presents such a refined repository $\mathcal{C}$. Native types, such as R, are intersected with semantic types, such as *Cel* or *Fh*, to specify meaning, e. g. in `cl2fh` the type R∩*Cel* describes a temperature represented by a real number w.r.t. the Celsius scale whereas the type R∩*Fh* describes a temperature represented by a real number w.r.t. the Fahrenheit scale. Type variables (α, α') facilitate generic combinators like `tmp` which is applicable to temperatures in both measurement systems.

We can now find a meaningful composition that returns a temperature in Fahrenheit by constructing the inhabitant `cl2fh (tmp (Tr O))` of the type R∩*Fh*,

```
O       : TrObj
Tr      : TrObj → D((R, R)∩Cart, R∩Gpst, R∩Cel)
pos     : D((R, R)∩α, R∩α′, R) → ((R, R)∩α, R∩α′)∩Pos
cdn     : ((R, R)∩α, R)∩Pos → (R, R)∩α
fst     : ((R, R)∩Coord → R)∩(Cart → Cx)∩(Polar → Radius)
snd     : ((R, R)∩Coord → R)∩(Cart → Cy)∩(Polar → Angle)
tmp     : D((R, R), R, R∩α) → R∩α
cc2pl   : (R, R)∩Cart → (R, R)∩Polar
cl2fh   : R∩Cel → R∩Fh
```

Fig. 3: Semantically refined repository $\mathcal{C}$

formally:

$$\mathcal{C} \vdash \texttt{cl2fh}\ (\texttt{tmp}\ (\texttt{Tr}\ \texttt{O})) : \texttt{R}\cap Fh$$

In general, the situation in combinatory logic synthesis is the following. We are *given* a repository of component names $\texttt{X}_\texttt{i}$ (regarded as combinator symbols in combinatory logic) with associated implementations C_i of type τ_i in a native implementation language L1

$$\texttt{X}_\texttt{1} \triangleq C_1 : \tau_1, \ldots, \texttt{X}_\texttt{n} \triangleq C_n : \tau_n$$

Thus, a combinator symbol $\texttt{X}_\texttt{i}$ is used as a placeholder for a concrete implementation C_i. In addition, we are *given* an associated repository as a combinatory type environment

$$\mathcal{C} = \{\texttt{X}_\texttt{1} : \phi_1, \ldots, \texttt{X}_\texttt{n} : \phi_n\}$$

where ϕ_i represents τ_i augmented by semantic information describing the type of the implementation of C_i in L1. Then, we *ask for* combinatory compositions e with $\mathcal{C} \vdash e : \phi$ such that e satisfies, in addition, the property of *implementation type correctness* requiring that e be a well-typed program in L1 after substituting all occurring combinator symbols $\texttt{X}_\texttt{i}$ with their corresponding implementations C_i.

Since repositories $(\Gamma, \mathcal{C})$ may change, we consider the *relativized inhabitation* problem: given $\mathcal{C}$ and ϕ, does there exist e such that $\mathcal{C} \vdash e : \phi$? Later on, we will use the abbreviating notation $\mathcal{C} \vdash ? : \phi$. In the tracking example above such an $e = \texttt{cl2fh}\ (\texttt{tmp}\ (\texttt{Tr}\ \texttt{O}))$ is the synthesized composition for the request $\phi = \texttt{R}\cap Fh$. Even in simple types, relativized inhabitation is undecidable (as explained in [1], this can be traced to the Linial-Post theorem [14]) and can be considered a Turing-complete logic programming language for generating compositions (see [1]). Here, $\mathcal{C}$ can be viewed as a logic program, the types of combinators $(X : \phi) \in \mathcal{C}$ are its rules, ϕ its input goal, and search for inhabitants its execution semantics.

2.3 Staged Extension

In order to flexibilize combinatory logic synthesis, staged composition synthesis (SCS) was proposed in [5]. SCS introduces a metalanguage, L2, in which L1-code

can be manipulated. The metalanguage is essentially the $\lambda_e^{\Box\rightarrow}$-calculus of Davies and Pfenning [15] which introduces a modal type operator, $\Box$, to inject L1-types into the type-language of L2. Intuitively, a type $\Box\tau$ can be understood to describe L1-code of L1-type τ. A second repository containing *composition components* with implementations in L2 is introduced. Then, synthesis automatically composes both L1- and L2-components, resulting in more flexible and powerful forms of composition since complex L1-code-maniualations, including substitutions of code into L1-templates, may be encapsulated in composition components. It is a nice consequence of the operational semantic theory of $\lambda_e^{\Box\rightarrow}$ that computation can be *staged*. For a composition e of type $\Box\tau$, it is guaranteed that all L2-operations can be computed away in a first *composition time* stage, leaving a well typed L1-program of type τ to be executed in a following *runtime* stage.

3 Combinatory Logic Synthesizer

In order to make composition synthesis feasible and accessible for experiments, and for application to realistic synthesis scenarios, we implemented the synthesis tool (CL)S. The core synthesis algorithm implements an optimized version of the inhabitation procedure [4] in Microsoft™ F# and C# using the Microsoft™ .NET-framework. In the following we discuss various features of (CL)S. Note also, that we extended (CL)S for conducting synthesis within SCS (cf. Sec. 2.3).

3.1 Tool

Input: To allow a user to specify component repositories, we defined an input language for (CL)S which is closely related to the mathematical notation of intersection types. To aid the user in specifying repositories (CL)S provides editor-extensions (for Microsoft™ Visual Studio 2013 and Notepad++) with syntax-highlighting, code completion, etc. For convenience and more concise representation of types we extended the input language by covariant type constructors, e.g., the type $(\mathtt{R},\mathtt{R})$ represents a pair of real numbers where $(\cdot,\cdot)$ is a type constructor for a pair of types. Furthermore, we allow variable kinding, where type variables, e.g. α in the example in Sect. 2.2, can be kinded by restricting the range of values (type constants) that the variable can be instantiated with, e.g., $\alpha_{\mathtt{temp}}{\sim}{>}\{\mathit{Cel},\mathit{Fh}\}$ states that $\alpha_{\mathtt{temp}}$ can be instantiated by the type constants *Cel* or *Fh* (but not by *Cart*). In order to represent taxonomic structures (cf. Sec. 2.2) on the semantic concepts describing intended usage of various components, (CL)S allows for explicit introduction of atomic subtyping constraints, e.g., $\mathtt{int} \leq \mathtt{R}$, expressing the fact that the type of integers is a subtype of the type of reals.

Processing: Processing of a synthesis question can be represented as a graph. This graph is called execution graph and contains two kinds of nodes. The first kind represents the choices of the algorithm and the other kind represents additionally generated synthesis tasks. Edges represent control flow. We implemented

a data structure for internal representation of the execution graph where the nodes are decorated with additional information concerning performance, processing, etc. In contrast to the algorithm presented in [4], which only decides if there is a solution to a given synthesis request, (CL)S also enumerates all (if the solution space is finite — otherwise the user is informed that cyclic solutions have been found) possible solutions. This last feature of (CL)S is essential concerning the applicability of composition synthesis.

(CL)S supports two different processing modes that have both been compiled for Windows as well as for Linux (using Mono). These are a batch-mode processing a synthesis request from a local file and a webservice-mode for synthesis requests on a remote server. The webservice exposes endpoints offering access via SOAP and REST. To this end, there are two hosting solutions of the webservice. First, there is a stand-alone server, mainly intended for usage in experiments. Second, there is a hosted version for application servers (IIS), intended for usage in industrial settings.

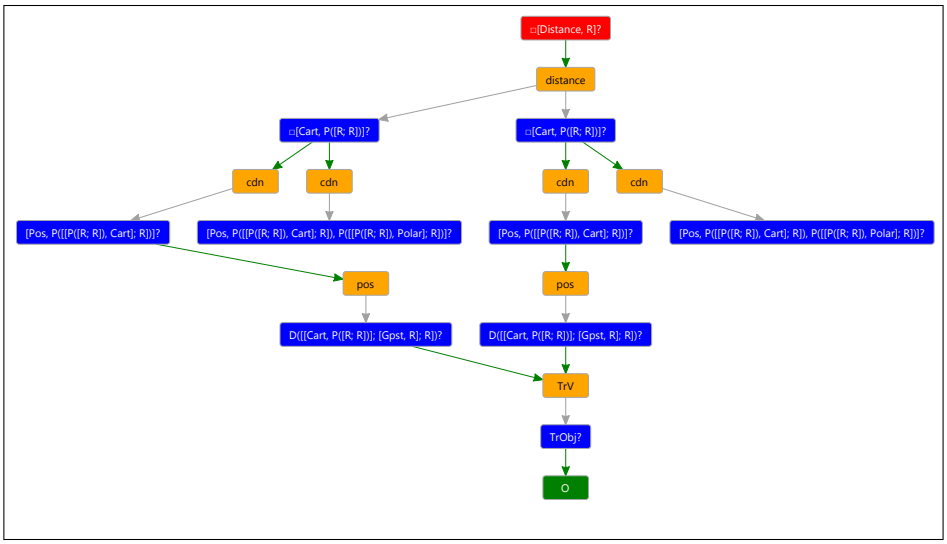

Fig. 4: Example of Execution Graph

Output: (CL)S provides various output formats depending on the intended usage of the results. First and foremost, there exists an XML-representation of the solutions to a given synthesis request for further processing. Also note that the XML-representation, among others, includes a human readable representation. Second, it is possible to export the execution graph corresponding to a given synthesis request to various formats, which can then be displayed visually (cf. Fig. 4). This feature is particularly important for an in depth analysis of a synthesis process. It supports both with regard to error-detection in the

design of repositories as well as for conducting experiments, allowing to apply graph-analysis and -processing tools.

Finally, various configurable logging functionalities are provided by (CL)S which is a crucial feature with regard to debugging and analyzing experiments.

3.2 Optimizations

To increase efficiency of the synthesis algorithm we implemented various optimizations that aim for acceleration of synthesis with respect to orthogonal aspects. First, in order to allow for scale-up and scale-out, we moved from a sequential implementation of the core synthesis algorithm to a concurrent one. Parallelization of the algorithm was necessary to exploit computation facilities in contemporary multi-core and cluster computing environments. Concurrency of the algorithm is facilitated by the above mentioned execution graph controlling synthesis. Parallelization needed sophisticated algorithm engineering as well as distributed techniques, like work-stealing queues and distributed message queues.

It has been shown that the simplest decision problems underlying combinatory logic synthesis (relativized inhabitation in bounded combinatory logic) is superexponential (EXPTIME-complete for monomorphic combinator types [3] and $(k+2)$-EXPTIME-complete for k-bounded combinatory logic [4]). Thus, heuristics for optimizing inhabitation is essential for application in practical scenarios. We implemented one such heuristic (which is based on the type theoretical problem of intersection type matching [16]) which formulates a necessary condition (referred to as "lookahead-strategy") for newly generated synthesis requests to be solvable. We experimentally compared the impact of the lookahead-strategy to the performance of the initial synthesis-algorithm of (CL)S. One experiment does arithmetic in $\mathbb{Z}_n$ and is parameterized by two integers, m and n (cf. Sec. 4.1 for details). Timing results for both implementations and different values of n and m can be found in Tab. 1 (cf. Sec. 4.1).

Using caches for successful and failed synthesis requests showed improvement. Cycle-detection in the execution graph allows for avoidance of cyclic inhabitants.

3.3 Architecture

The core architecture of (CL)S is depicted in Fig. 5. It is a modular master-slave architecture with a controller as master and thread-pooled slaves processing the synthesis requests with modular algorithms (e.g. the lookahead algorithm InhabOptimized). A work-stealing queue contains the synthesis tasks for each slave and the execution graph is the computation and synchronization data-structure. The success and fail caches are used by the controller for pruning unnecessary computations. The main challenges in design and implementation are concurrency issues imposed by the distribution of the algorithm.

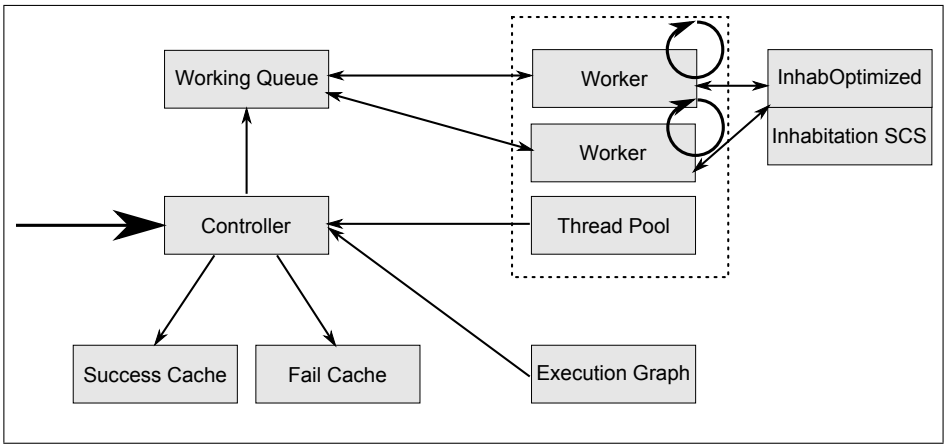

Fig. 5: Dependency Diagram for (CL)S

4 Current and Future Work

Currently, our (CL)S implementation is undergoing major structural changes. We integrate, on a more fundamental level, existing features that are currently added-on, such as type constants, variable kinding and covariant type constructors, and we tune the integration for performance.

In the current redesign cycle we emphasize maintainability of (CL)S and extend it by new features. From a user perspective, the most important new feature extends cycle detection to cycle analysis. In many applications, cyclic structures both on type- and on term level are essential for meaningful constructions. The current framework presents the whole solution space, including cycles, in a concise data structure. From a performance perspective, the single most important new feature extends the lookahead-strategy by utilizing inherent monotonicity of several inhabitation subroutines.

We simplify existing type theoretic algorithms and improve their cooperation by introducing pre- and postconditions to enhance overall efficiency of (CL)S. Additionally, we prove tight upper complexity bounds of several type theoretic problems that are utilized by the inhabitation procedure. During the analysis of underlying type theoretic problems, we identify combinatorial subroutines (e.g. minimal set cover in inhabitation) and separate them from the main algorithm. The new theoretical insights lead to a more profound understanding of inhabitation performance limits as well as improvement in specialized cases. The next step forward will be to develop a more systematic theory of optimization for composition synthesis.

In future work we plan to further extend the theoretic foundation of our framework needed to expand it into important application areas such as object-oriented code synthesis.

Although we work with complexity classes far above polynomial time, our approach has proven to be feasible in many practical scenarios of limited size.

Therefore, it is essential to analyze and understand the sources of complexity and non-determinism in (CL)S. In order to improve practical performance, we plan to algorithmically inspect and restructure user provided repositories to detect special cases. Furthermore, we see great potential in developing further heuristics to improve the lookahead-strategy.

4.1 Experimental Evaluation

In order to evaluate heuristic optimizations, we created a parameterized repository Γ_n^m for arithmetic in $\mathbb{Z}_n$, exploiting the fact that finite function tables can be coded by means of intersection types [2]. Γ_n^m consists of the identity-, successor- and predecessor-function in $\mathbb{Z}_n$ as well as an m-ary combinator to compose $\mathbb{Z}_n$-functions. For a $\mathbb{Z}_n$-function coded by an intersection type τ_n, we ask the inhabitation question $\Gamma_n^m \vdash ? : \tau_n$, i.e., we synthesize the particular $\mathbb{Z}_n$-function. The runtime performance of the initial (CL)S implementation, of a version of (CL)S using lookahead, and of the redesigned (CL)S are captured in Tab. 1. Practically infeasible tasks are marked with "–". The lookahead-strategy provides a considerable improvement over the initial implementation, making inhabitation questions that often appear in practice solvable. Further experience with the framework led to substantial performance gains with the redesigned (CL)S which allows for new practical applications.

(n, m)	Initial (CL)S	Lookahead-(CL)S	Redesigned (CL)S
$(2, 3)$	210 ms	111 ms	93 ms
$(3, 2)$	12504 ms	124 ms	98 ms
$(3, 3)$	–	354 ms	110 ms
$(4, 4)$	–	$7.5 * 10^6$ ms	121 ms
$(7, 7)$	–	–	1063 ms
$(10, 10)$	–	–	54250 ms
$(43, 3)$	–	–	8813 ms

Table 1. Experimental Runtime Performance for Γ_n^m

4.2 Interfaces and Language abstraction

The staged extension described in Sect. 2.3 is a valuable addition to our tool. Thus, it is an important part of our work to find encodings of L1-languages which are suitable for synthesis using L2-implementations. Our current language for L2-implementations is the $\lambda_e^{\square\rightarrow}$-calculus extended with a string based templating mechanism. We treat L1-code fragments as strings and allow L2-implementations to concatenate them and to fill in variables. After reduction no template variables are left and the resulting string is embedded into the XML reply format. This approach is easy to understand and very powerful, but it can be desirable to use tools more specialized for the structure of a given L1-language. Practical

applications show the diversity of possible L1-targets. They include classical programming languages like F# or Java, workflow description languages like BPMN and even highly specialized domain specific languages for tasks such as the configuration of nodes in a cloud computing environment. To gain a more structured view than the one provided by plain strings, without losing generality, we plan to change our templating system to use XML encodings of abstract L1-syntax trees. This would not only allow to reason about reusable combinators for tree transformations, but also enables a standardized interface for postprocessors. These can parse the synthesized XML code fragment and translate it to any desirable representation. They can be implemented in any high level programming language and make use of the best available tools for the task at hand. The XML encoding would allow to incorporate a limited form of static model checking directly into the synthesis process: additionally to type checking, XPath could be used to assert structural properties of the generated L1-code and to sort out unwanted solutions early.

Figure 6 summarizes our envisioned synthesis pipeline. It starts with users providing source code fragments, possibly even as diagrams. These fragments are then (semi-automatically) translated to typed combinators with XML fragments and templating abstractions over XML fragments as implementations. Types (depicted as puzzle pieces for L1 and puzzle pieces with holes for L2) are intersected with user provided semantic types that guide the inhabitation process (depicted as colors). The typed combinator repository is then used as input for SCS. Combinators for L2 are reduced and the result will be XML fragments describing the abstract syntax tree of the synthesized programs. In a final step those syntax trees are postprocessed to real programs or possibly graphical representations of real programs. Pre- and postprocessing steps require careful design tailored for each individual inhabitation language. They build the bridge to the problem independent interface exposed by our inhabitation web service depicted as the SCS screw driver tool.

4.3 Software engineering process

The software engineering process plays a major role in the design of a complex tool like (CL)S. Currently, we mainly develop research prototypes with a focus on understandable and correct implementations demonstrating our theoretical results. In future, we plan to introduce a more formal development model in order to facilitate the growth of our team and to react to the short periods in which student members join us for their own project work. To this end, we plan to turn our research demonstrations into integration tests, which are automatically executed by a continuous integration server. This also involves an automatic build and deployment process, which has the additional benefit of simplifying cloud installations. Furthermore, we are currently reevaluating existing and creating new guidelines for our code structure. We explicitly leave open release and versioning policies to be able to catch up with new research results.

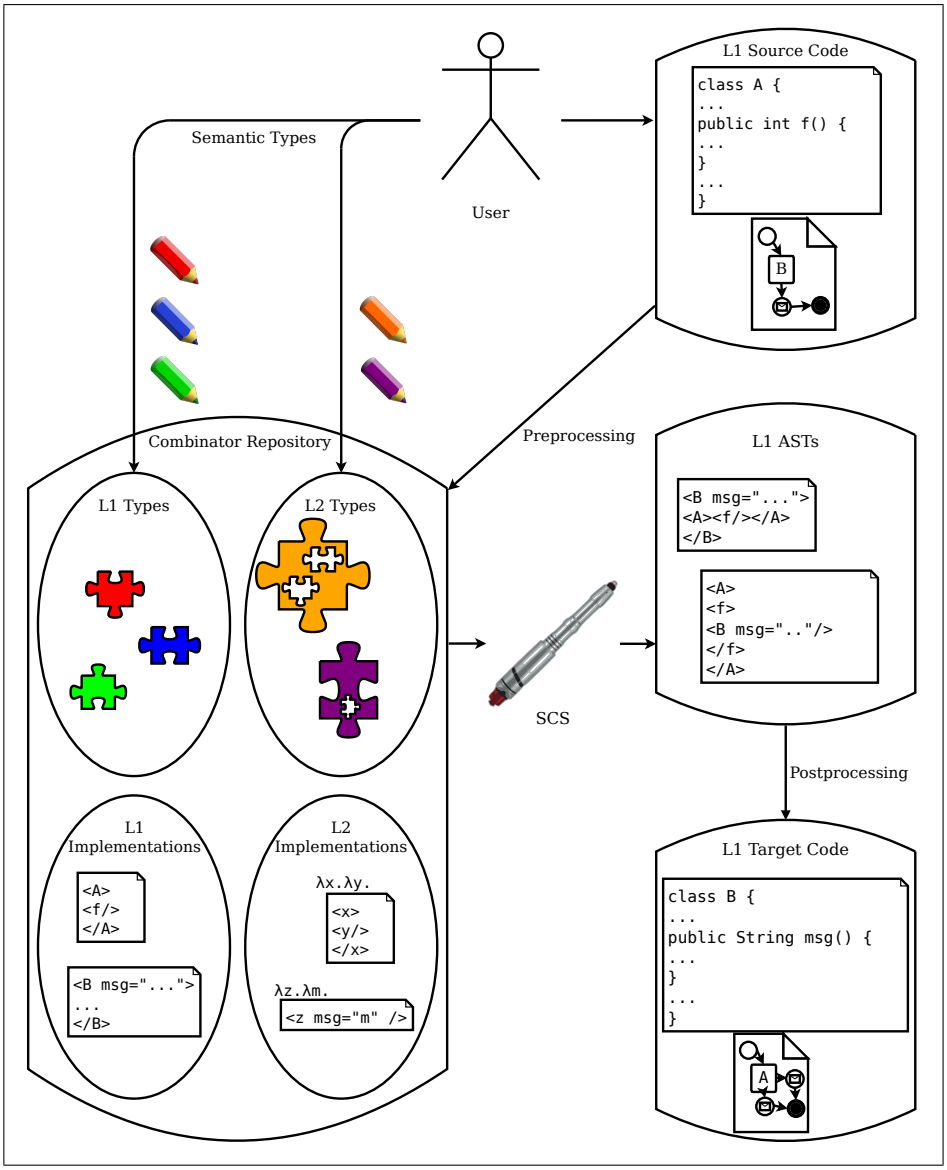

Fig. 6: Visualization of our envisioned process pipeline

5 Applications

(CL)S has already been applied to a variety of synthesis scenarios.

1. ArchiType[3], a rapid-prototyping tool for software architects, uses (CL)S for synthesizing software connectors in software architectures from a repository of connector components. ArchiType has been used in synthesizing and generating C#-code for an enterprise resource planning system and for an eCommerce system. These are relatively large-scale and realistic scenarios.
2. Combinatory process synthesis [17] uses (CL)S (with SCS) to generate deployable BPMN 2.0 workflows from a repository of process components.
3. (CL)S has also been used to synthesize control programs for LEGO® NXT robots from a repository of atomic and complex control components [18].
4. In [19], (CL)S has been used to synthesize configurations for virtual machine images in cloud computing (OpenNebula) and corresponding deployment code for instantiating these images from a repository containing various configuration components and comprehensive IT infrastructure information.

As a final illustration we discuss in more detail one particular synthesis scenario to which (CL)S (with SCS) has been applied. The Java Virtual Machine (JVM) is host to various languages which are of interest as synthesis targets (`L1`). We have successfully generated Dependency Injection configurations for the Spring framework [20] and a current research focus is to support synthesis with Java mixins [21]. The importance of JVM languages led to the design of a Scala based framework to programmatically generate repositories from reflected Java classes. This framework provides an embedded domain specific language (EDSL) designed as an extensible abstract interpreter. The EDSL exposes two syntactic views on the specification of repositories. For use in algorithms all specification components (e.g., arrows in type signatures) are available as verbosely named methods, adding the respective component to the repository which is currently built. To describe a repository in a human readable form, the EDSL also exposes a syntactic view, which closely resembles the mathematical specification language. Interpretation results are calls to the (CL)S webservice. They trigger inhabitation requests using the specified repository and goal type. Scala allows to seamlessly interoperate with other JVM languages and its expressive type system can detect many specification errors at compiletime, even prior to the creation of a specific repository. We aim to provide practically usable interfaces for programmers not familiar with type theory. Listing 1 shows the interface to be implemented to create a synthesis request for a Dependency Injection candidate. When passed to our Scala based preprocessing framework class representations returned by the methods of an instance of this interface are analyzed via reflection. Using the repository specification EDSL, typed combinators are created for each existing object in the library context and for all constructor-, setter- and factory-methods of the library classes. For example, a class `EmployeeController`

[3] `http://www-seal.cs.tu-dortmund.de/seal/pages/research/architype_en.shtml`

with a constructor requiring a data access object (DAO) to obtain information about employees, triggers the creation of the following combinator

$$\texttt{EmployeeController} : (\Box\texttt{Scope} \cap \alpha) \to \Box(\texttt{EmployeeDAO} \cap \alpha) \to \Box(\texttt{EmployeeController} \cap \alpha \cap \texttt{setterTarget})$$

$$\begin{aligned}\texttt{EmployeeController} : \lambda&\texttt{Scope}.\ \text{letbox}\ \texttt{scope} = \texttt{Scope}\ \text{in}\\ \lambda&\texttt{DAO}.\ \text{letbox}\ \texttt{dao} = \texttt{DAO}\ \text{in}\\ &\text{box}\ \texttt{"<bean xsi:type='cls:Constructor'>"}\\ &\quad \texttt{"<name>EmployeeController</name>"}\\ &\quad \texttt{"<typeName>EmployeeController</typeName>"}\\ &\quad \texttt{"<scope>"}\texttt{scope}\texttt{"</scope>"}\\ &\quad \texttt{"<argument>"}\\ &\quad \texttt{"<typeName>EmployeeDAO</typeName>"}\\ &\quad \texttt{"<reference>"}\texttt{dao}\texttt{"</reference>"}\\ &\quad \texttt{"</argument>"}\\ &\quad \texttt{"</bean>"}\end{aligned}$$

```
1 public interface InhabitationRequest {
2     public Class [] libraryClasses();
3     public ConfigurableApplicationContext libraryContext();
4     public String classNameExclusionRegexp();
5     public Class targetType();
6     public String targetScope();
7 }
```

Listing 1. Interface to request Spring Dependency Injection candidates

The combinator takes two arguments, a scope supplying lifecycle information to Spring, and the data access object. It operates on boxed L1-terms, where L1 is an XML based representation of Spring Beans translatable to a Dependency Injection configuration via postprocessing. Its result is an L1-term describing how to instantiate an `EmployeeController`. Semantic constant `setterTarget` in the combinator result type indicates that setter-methods of the object may be used to inject more dependencies into it. The above mentioned extended $\lambda_e^{\Box\to}$-implementation already creates XML, but still treats it as strings, making it a candidate for the planned enhancements described in Sect. 4.2. Synthesis results for the described scenario and two given existing database connections are available online[4].

Compared to earlier versions [20], our recent developments allow more succinct combinator types, because cyclic scenarios can be addressed within the algorithm. Repository creation as well as calls to the webservice are automatized and wrapped into a Maven[5] plug-in, enabling the synthesis of Spring configurations at the click of a mouse button.

[4] http://www-seal.cs.tu-dortmund.de/seal/pages/research/DI-example.zip
[5] http://maven.apache.org/

References

1. Rehof, J.: Towards Combinatory Logic Synthesis. In: BEAT'13, 1st International Workshop on Behavioural Types, ACM (January 22 2013)
2. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A Filter Lambda Model and the Completeness of Type Assignment. Journal of Symbolic Logic **48**(4) (1983) 931–940
3. Rehof, J., Urzyczyn, P.: Finite Combinatory Logic with Intersection Types. In: Proceedings of TLCA'11. Volume 6690 of LNCS., Springer (2011) 169–183
4. Düdder, B., Martens, M., Rehof, J., Urzyczyn, P.: Bounded Combinatory Logic. In: Proceedings of CSL'12. Volume 16 of LIPIcs., Schloss Dagstuhl (2012) 243–258
5. Düdder, B., Martens, M., Rehof, J.: Staged Composition Synthesis. In: Proceedings of ESOP'14. Volume 8410 of LNCS., Springer (2014) 67–86
6. Manna, Z., Waldinger, R.: Fundamentals Of Deductive Program Synthesis. IEEE Transactions on Software Engineering **18** (1992) 674–704
7. Bodik, R., Jobstmann, B.: Algorithmic Program Synthesis: Introduction. International Journal on Software Tools for Technology Transfer **15**(5-6) (2013) 397–411
8. Lustig, Y., Vardi, M.Y.: Synthesis from Component Libraries. In: FOSSACS. Volume 5504 of LNCS., Springer (2009) 395–409
9. Hindley, J.R., Seldin, J.P.: Lambda-calculus and Combinators, an Introduction. Cambridge University Press (2008)
10. Steffen, B., Margaria, T., von der Beeck, M.: Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach. In: In ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS97). (1997)
11. Haack, C., Howard, B., Stoughton, A., Wells, J.B.: Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In: AMAST. Volume 2422 of LNCS., Springer (2002) 83–98
12. Wells, J.B., Yakobowski, B.: Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis. In: LOPSTR 2004. Volume 3573 of LNCS., Springer (2005) 262–277
13. Freeman, T., Pfenning, F.: Refinement Types for ML. In: Proceedings of PLDI'91, ACM (1991) 268–277
14. Linial, L., Post, E.L.: Recursive Unsolvability of the Deducibility, Tarski's Completeness and Independence of Axioms Problems of Propositional Calculus. Bulletin of the American Mathematical Society **55** (1949) 50
15. Davies, R., Pfenning, F.: A Modal Analysis of Staged Computation. Journal of the ACM **48**(3) (2001) 555–604
16. Düdder, B., Martens, M., Rehof, J.: Intersection Type Matching with Subtyping. In: Proceedings of TLCA'13. Volume 7941 of LNCS., Springer (2013)
17. Vasileva, A.: Synthese von Orchestrationscode für Cloud-basierte Dienste. Diploma thesis, Technical University of Dortmund, Department of Computer Science (2013)
18. Wolf, P.: Entwicklung einer Adapters mit VI Scripting (LabVIEW) zur Synthese von LEGO® NXT-VIs aus einem Repository. Bachelor's thesis, Technical University of Dortmund, Department of Computer Science (2013)
19. Plate, S.: Automatische Generierung einer Konfiguration für virtuelle Maschinen unter Zuhilfenahme eines Inhabitationsalgorithmus. Bachelor's thesis, Technical University of Dortmund, Department of Computer Science (2013)
20. Bessai, J.: Synthesizing Dependency Injection Configurations for the Spring Framework. Master's thesis, TU Dortmund, Department of Computer Science (2013)
21. Bracha, G.: The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, University of Utah, Salt Lake City, UT, USA (1992)