User Support for the Combinator Logic Synthesizer Framework

Jan Bessai Anna Vasileva
Technical University of Dortmund, Germany
jan.bessai@tu-dortmund.de anna.vasileva@tu-dortmund.de

Usability is crucial for the adoption of software development technologies. This is especially true in development stages, where build processes fail, because software is not yet complete or was incompletely modified. We present early work that aims to improve usability of the Combinatory Logic Synthesizer (CL)S framework, especially in these stages. (CL)S is a publicly available type-based development tool for the automatic composition of software components from a user-specified repository. It provides an implementation of a type inhabitation algorithm for Combinatory Logic with intersection types, which is fully integrated into the Scala programming language. Here, we specifically focus on building a web-based IDE to make potentially incomplete or erroneous input specifications for and decisions of the algorithm understandable for non-experts. A main aspect of this is providing graphical representations illustrating the step-wise search process of the algorithm. We also provide a detailed discussion of possible future work to further improve the understandability of these representations.

1 Introduction

The Combinatory Logic Synthesizer (CL)S Framework provides a publicly available [8] development tool, which is fully integrated into the Scala programming language and can automatically compose software based on types. Type specifications for (CL)S are based on Combinatory Logic with intersection types [15] and automatic software composition is performed by answering the type inhabitation question: $\Gamma \vdash ? : \tau$. In words this question reads: given a set of typed combinators, where each combinator represents a software component, find all applicative terms formed from the combinators in Γ, which have type τ. Integration into Scala allows to reuse programming skills and greatly simplifies the specification of combinators. While user-input is easy to provide and any Scala IDE can be used to program and manipulate combinators, it can be difficult to understand why the algorithm involved in solving the type inhabitation question does or does not produce certain expected solutions. This is especially true, when (CL)S is used to automatically compose non-trivial software from large component bases, which is its main use-case [6, 7, 9, 16, 25].

According to ISO 9241-11:2018 [28], the definition of usability is "extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". In our case, the system, (CL)S, is intended to be used by normal programmers, who are not experts on type theory, and want automatic composition to enhance their efficiency in the context of large collections of software components without being dissatisfied by inexplicable (non-)solutions. To achieve this aim, a web-based IDE for debugging and improving type specifications is being developed. It will especially support programmers in development stages, where type specifications are still incomplete, causing the

P. Masci, R. Monahan, V. Prevosto (Eds.): Formal Integrated Development Environment 2018 (F-IDE 2018).
EPTCS 284, 2018, pp. 16–25, doi:10.4204/EPTCS.284.2

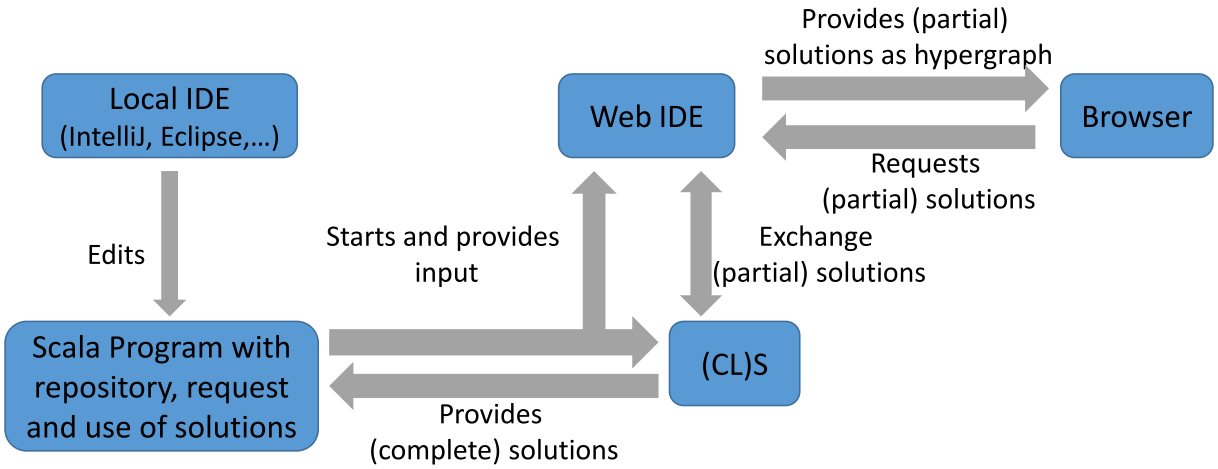
algorithm to find not enough or too many solutions. An important aspect of this is to visualize the search process performed by the inhabitation algorithm, as well as its – potentially infinite – solutions spaces. The main contributions of this paper are: to provide an IDE-based view-point on type inhabitation, to investigate how hypergraphs are useful in this context, and to pose a number of interesting research questions. The paper is organized as follows: in the remainder of this section we will discuss some related work. Section 2 includes a brief presentation of the (CL)S framework, architecture level aspects of connecting it to an IDE, and more context on the running example of movement combinators. In Section 3 we discuss the formalism of tree grammars in context of type inhabitation, their relation to hypergraphs and how the IDE presents them to users. Finally, in Section 4 we summarize and pose some interesting research questions for future work.

1.1 Related Work

Usability is well-studied and its various aspects [34] also include the tools for implementing software [33]. Here, we focus on the implementation phase of projects using the (CL)S framework. A similar effort in the setting of program synthesis and verification has been undertaken within the Leon project [10], which provides a web-based IDE [3]. Leon uses several SMT-solvers as its back-end. It performs verification on the level of individual Scala AST nodes and synthesis is based on invariants. In contrast, (CL)S has a single back-end algorithm that synthesizes function compositions from type specifications. The Leon IDE is text-oriented and shows results next to individual lines of code, while graphical result representations are central to our development. For users preferring text, we annotate each node representing a combinator with source positions indicating its point of origin. In the area of automatic verification, the Why3 IDE [11] is similar to Leon and also textually links the output of various SMT solvers to positions in code. The RESOLVE programming language [29], Coq [30, 4] and the LEAN theorem prover [20] have similar IDEs to support (semi-)manually solved proof obligations for verified programs. Resembling our approach and unlike the aforementioned, the Globular proof assistant [5] helps to build proofs graphically. The graphical representation in Globular is based on category theory string diagrams and users manipulate graphs directly, while the hypergraphs in (CL)S are automatically generated. Globular requires users to be experts in category theory, while (CL)S aims to not require expertise on type-theory. IDE support is crucial in the complicated process of program verification, and the list of such developments is too long to fully enumerate. The above examples are selected for being web-based. In contrast to most web-based IDEs, we do not try to relocate source-code development into the browser, but rather focus on graphically assisting it. This means, developers can continue using the Scala IDE they are used to. From the web-based approach we gain platform independence. At the time of writing, most native client-side user interface libraries require platform specific code or application setup instructions. Our development works out-of-the-box with a browser and sbt [2], which is necessary for using (CL)S even without the IDE.

We hope that our work provides useful insights for building IDEs for other settings. It could especially help with Petri net and hypergraph based synthesis [21]. To our knowledge, there is no other tool to debug intersection type specifications [14], which are an important area of research [36, 27] with multiple recent applications to synthesis [18, 12, 22] beyond Combinatory Logic [15].

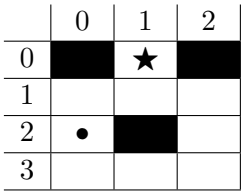
2 An Overview of (CL)S Scala Framework



The illustration above provides an overview of the data flow when using and debugging specifications in (CL)S. Programmers specify and implement combinators in Scala using any local IDE. Their programs also include synthesis targets and put results of the algorithm to further use, e.g. by interpreting combinator applications as function calls. The framework itself can be used as a library from any Scala program. If the additional debugging capabilities of the web-based IDE are desired, it is started by instantiating a controller for the Play web framework [32]. Starting the application will instantiate a web server, which hosts a web site where users can access debugging information, e.g. in form of hypergraph-based visualizations of the search process. Behind the scenes, the debugger communicates with the algorithm implementation provided by (CL)S, which has appropriate interfaces to observe its internal state.

There are four basic rules to control the process of type inhabitation [15]. The first rule (if $c : \tau \in \Gamma$ then $\Gamma \vdash c : S(\tau)$) allows to use any combinator c present in repository Γ with type τ and to assume that it has type $S(\tau)$, where S is a substitution mapping type variables in τ to types without variables. This way, e.g. the identity function can have type $\alpha \to \alpha$ that can turn into $Int \to Int$ or $String \to String$, depending on the choice of substitution. In order to make type inhabitation decidable, substitutions are drawn from a finite space (instead of guessed), which is part of the input specification. The second rule (if $\Gamma \vdash M : \sigma \to \tau$ and $\Gamma \vdash N : \sigma$ then $\Gamma \vdash MN : \tau$) allows to apply combinators with function types to appropriately typed arguments. Moreover, we have the intersection introduction rule ($\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$ implies $\Gamma \vdash M : \sigma \cap \tau$) to type the same term with two types, if both types can be derived. Lastly, the type system supports subtyping ($\Gamma \vdash M : \sigma$ and $\sigma \leq \tau$ implies $\Gamma \vdash M : \tau$). When answering an inhabitation question, (CL)S will search for combinators and their arguments, such that synthesized applicative terms are well-typed. Input specifications are the combinator repository, allowed substitutions for type variables (if there are any), a subtype relation on type constants to define $\leq$, and the requested type.

We consider labyrinths as an example and try to find all paths from an entrance to a goal. The following labyrinth has start position $(0,2)$ and goal position $(1,0)$:



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

There are multiple possible paths to reach the goal position. Repository Γ_{ex} shows the labyrinth in mathematical notation. Each entry in Γ_{ex} consists of a combinator name and its type description. The repository represents the start position and all possible one-step moves as typed combinators. Types indicate column and row positions in the labyrinth. Combinators for going up, down, left or right are functions from a start to a destination position. Intersection types allow movement combinators to have multiple types at once, e.g. combinator *up* can be used to go from $Pos(1,1)$ to $Pos(1,0)$ and $Pos(2,2)$ to $Pos(2,1)$. To make the example more readable, we avoid variables, which would have allowed for specifications like $up : Pos(\alpha, PlusOne(\beta)) \to Pos(\alpha,\beta)$. We get all possible paths through the labyrinth by asking for the goal position, e.g. $\Gamma_{ex} \vdash ? : Pos(1,0)$. The algorithm computes all solutions in form of tree grammars which will be shown as hypergraphs.

3 IDE for the (CL)S Framework

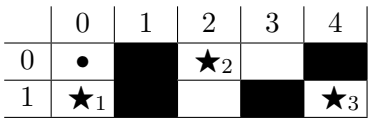
The (CL)S framework recursively grows the set of production rules of a tree grammar to describe solutions. Tree grammars are well-known from literature [13, 24] and we consider the generalized case of regular tree grammars without restrictions on the arity of terminal symbols. Formally we have:

Definition 1 *A tree grammar G is a 4-tuple $(S,N,\mathcal{F},R)$ with a start symbol $S \in N$, a set N of nonterminals, a set $\mathcal{F}$ of terminal symbols, and a set R of productions rules of form $\alpha \mapsto f(\beta_1,\beta_2,\ldots\beta_n)$, where $n \geq 0$, $\alpha,\beta_1,\beta_2,\ldots,\beta_n \in N$ are nonterminal and $f \in \mathcal{F}$ is terminal. For a given tree grammar $G = (S,N,\mathcal{F},R)$ and nonterminal $\alpha \in N$, $\mathcal{L}_\alpha(G)$ is the least set closed under the rule: if $\alpha \mapsto f(\beta_1,\beta_2,\ldots,\beta_n) \in R$ and for all $1 \leq k \leq n : t_k \in \mathcal{L}_{\beta_k}(G)$ then*

$$f(t_1,t_2,\ldots,t_n) \in \mathcal{L}_\alpha(G).$$

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

Solutions in (CL)S are terms M, which are well-typed for a requested type τ relative to the type assumptions explained above. Given Γ and τ, (CL)S constructs a tree grammar $G = (\tau,N,\mathcal{F},R)$ such that $\tau \in N$ and for all $\sigma \in N$ we have $M \in \mathcal{L}_\sigma(G)$ if and only if $\Gamma \vdash M : \sigma$. In other words, we get a tree grammar where right hand sides of rules start with a combinator symbol followed by the types of arguments required to obtain the type on the left hand side of the rule by applying the combinator. The start symbol is the user requested target type.
Let us consider the following example labyrinth to illustrate the search process:



$$\Gamma = \{ up : (Pos(0,1) \to Pos(0,0)) \cap (Pos(2,1) \to Pos(2,0)),$$
$$down : (Pos(0,0) \to Pos(0,1)) \cap (Pos(2,0) \to Pos(2,1)),$$
$$left : Pos(3,0) \to Pos(2,0),$$
$$right : Pos(2,0) \to Pos(3,0),\ start :\ Pos(0,0) \quad \}$$

For goal position $\bigstar_1$, we ask $\Gamma \vdash ? : Pos(0,1)$ and combinator *down* can be used with argument $Pos(0,0)$. The first computed tree grammar entry will be $Pos(0,1) \mapsto down(Pos(0,\omega) \cap Pos(\omega,0))$. For internal implementation reasons of the search procedure, (CL)S chooses the nonterminal for the argument of down to represent a type, which is subtype equal to $Pos(0,0)$ (in Combinatory Logic with intersection types we have: $Pos(0,\omega) \cap Pos(\omega,0) \leq Pos(0,0) \leq Pos(0,\omega) \cap Pos(\omega,0)$, because type constructors like Pos are co-variant in their arguments, distribute over intersection and ω is the universal supertype of every other type). Type $Pos(0,\omega) \cap Pos(\omega,0)$ will become the next target, for which two rules are computed: $Pos(0,\omega) \cap Pos(\omega,0) \mapsto start()$, which is obvious, and $Pos(0,\omega) \cap Pos(\omega,0) \mapsto up(Pos(0,1))$, which is perhaps surprising. The computed tree grammar is not only sound (its words are well-typed terms), but also complete (all requested well typed terms are words). Hence, the cyclic second rule causes terms like $down(up(down(start))), down(up(down(up(down(start)))))$, ... to be derivable. Inhabitation stops, because a rule for the argument of *up* can be found in the already computed grammar ($Pos(0,1)$ is the left hand side of the first rule) and no further recursive targets exist. For goal position $\bigstar_2$ the algorithm computes rules $R = \{Pos(2,0) \mapsto up(Pos(2,\omega) \cap Pos(\omega,1)),$ $Pos(2,0) \mapsto left(Pos(3,\omega) \cap Pos(\omega,0)),\ Pos(2,\omega) \cap Pos(\omega,1) \mapsto down(Pos(2,0)),\ Pos(3,\omega) \cap Pos(\omega,0) \mapsto right(Pos(2,0))\}$. This time rules are cyclic and unproductive, no word can be derived for the start symbol $Pos(2,0)$. In the implementation, all unproductive rules are pruned from the grammar and one motivation for having a debugger is to inform users about that process. Another motivation arises when considering goal $\bigstar_3$, for which no combinator exits. The tree grammar will be empty and users would have to check the entire repository to find out why.

We introduce the *hypergraphs* [19] as a graphical representation of tree grammars.

Definition 2 *A directed labeled hypergraph H over an alphabet $\mathcal{F}$ is a 5-tuple H = (V, E, nod_E, nod_V, lab) where V is a finite set of nodes, E is a finite set of hyperedges, incidence is specified by a function $nod_E : E \to V^*$ and a relation $nod_V \subseteq V \times E$, and labels are given by a function lab: $E \to \mathcal{F}$.*

Every edge in a hypergraph has outgoing connections described by nod_E and incoming connections described by nod_V. Outgoing connections are finite vectors of nodes. This is different from normal graphs, where edges only connect two nodes. All tree grammar nonterminals are represented by nodes ($V = N$). For each production $\alpha \mapsto f(\beta_1, \beta_2, \ldots \beta_n)$ we add an edge e with $\mathrm{nod}_V(\alpha) = e$ and $(e, (\beta_1, \beta_2, \ldots \beta_n)) \in \mathrm{nod}_E$ where $\mathrm{lab}(e) = f$.

Figures 1 and 2 provide an overview of the hypergraphs for the tree grammars for the prior labyrinth example. The hypergraph for position $\bigstar_1 = Pos(0,1)$ in Fig. 1 contains nodes for every nonterminal (type) and combinator usages are modelled by edges. Outgoing edge connections are numbered to indicated argument positions. We draw nodes and edges using boxes and circles. The cycle between combinators *up* and *down* is immediately visible in Fig. 1. We can escape the cycle using edge *start*, which has no outgoing connections and thereby models use of a combinator not depending on additional recursively synthesized targets. The labels collected for all ways through the hypergraph from *start* to $Pos(0,1)$ are words of the grammar and valid movement instructions through the labyrinth. For $\Gamma \vdash ? : Pos(2,0)$, the IDE shows a message

that there is no solution. In this case, the user can comprehend the problem by means of the visualization provided by the debugger mode. Figure 2 shows the cyclic rule obtained in the last step. By comparing the graphs, we see that this hypergraph is different form the first one (Fig. 1): there is no edge without outgoing connections to break cycles. In the visualization, all edges involved in unbreakable cycles are marked in red.

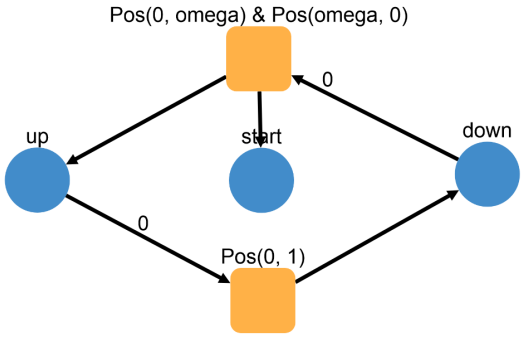


Figure 1: $Pos(0,1)$

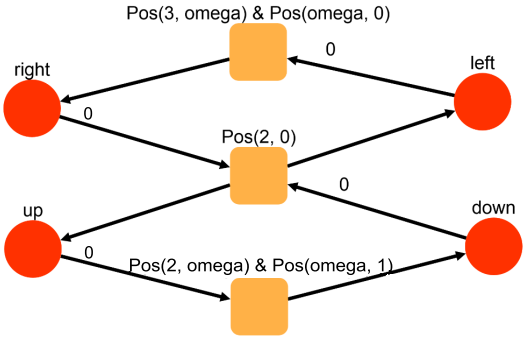


Figure 2: $Pos(2,0)$

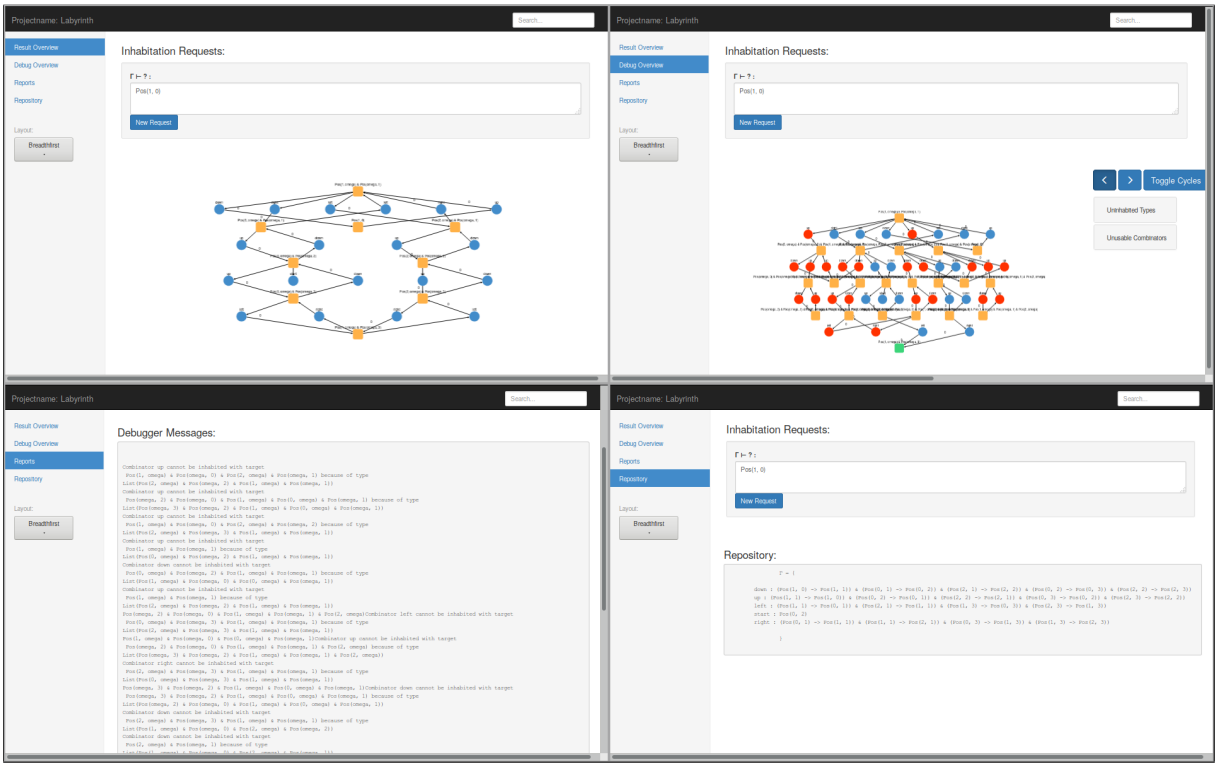


Figure 3: Debug Overview

The web-based IDE provides four different perspectives (Fig. 3). The *Result Overview* presents the solutions of the inhabitation problem in form of a hypergraph. In order to inform about unsuccessful inhabitation, a message is shown instead of the empty hypergraph. The *Debug Overview* (Fig. 3, top right) and *Reports* (Fig. 3, bottom right) perspectives pro-

vide detailed graphical and textual information about the inhabitation process, which otherwise can be unexpected and incomprehensible. Users can access their specification in the *Repository* (bottom right) perspective. The *Result Overview* perspective also provides the possibility to make a new request by means of the browser-based IDE. This enables fast user interactive experiments with types different (e.g. more specific or generic) from the programmatic request stated in Scala. For the visualization of the hypergraph construction, we use the open-source JavaScript library Cytoscape [23], which facilitates the fast and interactive representation of hypergraphs exchanged in a simple JSON format. User interactiveness allows to zoom into a graph as well as to move the nodes and edges. Layout choices are supported by eight different automatic layout algorithms [23]. We use the Bootstrap HTML components [1] to gain platform and browser independence. In the *Debugger Overview* perspective, users can see the generation of solutions in a step-wise process. Figure 3 (top right) shows a step of the construction process of the solution for the labyrinth example Γ_{ex} presented in Section 2. In the current step, we see that there is also an unproductive cycle. Figure 4 shows a zoomed-in and manually re-layouted part of the graph with a cycle in this step. Because of the intersection type specific rules, combinator *up* can be used with type $(Pos(\omega,3) \cap Pos(\omega,2) \cap Pos(1,\omega) \cap Pos(0,\omega) \cap Pos(\omega,1)) \to (Pos(0,\omega) \cap Pos(\omega,2) \cap Pos(1,\omega) \cap Pos(\omega,1))$. This is surprising for non-experts. However, it does not lead to invalid solutions, because all possible inhabitants for the argument type of *left* are generated from an unproductive cycle. We include a button to toggle all unproductive cycles providing a clean unsurprising view. There are no combinators for goal $\bigstar_3$, therefore the associated hypergraph is uninformative. The graph in the *Debug Overview* perspective contains only type $Pos(4,1)$ as a green node, with color green indicating yet to-do recursive targets. Since there exists no suitable combinator in Γ, the graph for the next step is empty. Moreover, users can find information in the *Reports* perspective. It includes textual information about each uninhabited type encountered during the search process. Reasons for non-inhabitation (unproductive cycles, no usable combinators) are distinguished. The *Repository* perspective gives users an overview of their repository specifications. These are not always entered manually, but can also be programmatically constructed from a problem domain. In case of the labyrinth examples, it is easy to write a Scala program to create Γ from a two dimensional boolean array.

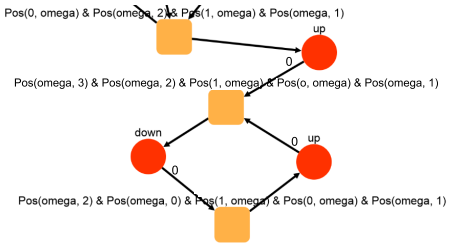


Figure 4: Unproductive cycle in solution construction

4 Conclusion and Future Work

The presented IDE for (CL)S is geared toward debugging incomplete or erroneous input type specifications. We provided an overview of the framework and IDE together with an easy to understand application example. The example is then used to illustrate the connection between

tree grammars generated by (CL)S and hypergraphs, which are used for visualization in the debugger. We have shown how intermediate synthesis steps can be visualized and unproductive cycles can be seen in the hypergraph. This step-wise visual construction of the hypergraphs may help non-experts understand the decisions of the algorithm. There are numerous areas of future work:

Evaluation While the debugger helped us to understand and debug our own examples, more evaluation is needed to see if it really helps non-experts. A possible evaluation scenario would include student groups and measure their effectiveness in solving a given task [35]. It would also be important to see which further features would be needed to make (CL)S scale to development teams. This is especially interesting, when developers try to understand type-specifications devised by others. There is little to no work on studying this subject in the area of software synthesis in general.

Performance While writing this paper we were able to greatly improve the performance of (CL)S, because the hypergraphs constructed for the labyrinth example revealed the generation of redundant recursive inhabitation targets. Due to the computational complexity of type inhabitation in Combinatory Logic with intersection types, which is above EXPTIME [15], we expect that there always will be scenarios where users have to wait for results. Some further optimizations have been present in an earlier F# based version of (CL)S, and just need to be ported. For drawing and layouting of hypergraphs, Cytoscape was fast enough for user interactive operation, but we expect limitations when solutions get too big. In this case, partial collapsing of hypergraphs might help.

Input specification quality Input specifications can include badly designed combinators. Indications of bad design would be unusable combinators, unnecessarily generic or overly specific types, or overly long parameter lists leading to implementation code smells. We plan to improve our IDE to provide user feedback by statically analyzing the repository. This kind of analysis will potentially give further insights on intersection types, relying on practical applications of techniques such as intersection type matching or unification [17]. It might lead to a formal understanding of what it means to be a component suitable for synthesis and reveal a potential connection to clean code in regular programming.

Algorithmic Artifacts We have seen that the search procedure will introduce hard to understand type artifacts, representing type $Pos(0,0)$ as $Pos(0,\omega) \cap Pos(\omega,0)$. The cycle shown in Fig.4 is such an artifact, because its types were created by the intersection introduction rule, which can have surprising consequences in the case of large intersections. A detailed case-to-case analysis of intersection types in practical experiments will be necessary to find good countermeasures to the aforementioned problems. Most other synthesis techniques rely on some form of intermediate representation and clearly more research is needed on how to make the connection of these representations to the initially specified problem obvious to non-experts.

In addition to these future work topics the structure of Petri nets can be represented as directed hypergraphs [31] and thus the iterative construction in [21] could be adapted to be shown in our IDE. In reverse, recently developed web-based tools for debugging and benchmarking Petri nets [26] have the potential to give useful input for our future work.

References

[1] Bootstrap. `https://getbootstrap.com/`, accessed: 2018-04-24

[2] Scala Build Tool (SBT). https://www.scala-sbt.org/, accessed: 2018-05-23

[3] Antognini, M., Blanc, R., Gruetter, S., Hupel, L., Kneuss, E., Koukoutos, M., Kuncak, V., Madhavan, R., Stucki, S., Suter, P.: Leon System for Verification, Synthesis and Repair, http://leon.epfl.ch/, accessed: 2018-04-27

[4] Arias, E.J.G., Pin, B., Jouvelot, P.: jsCoq: Towards Hybrid Theorem Proving Interfaces. In: Proceedings of the 12th Workshop on User Interfaces for Theorem Provers. pp. 15–27 (2016), https://doi.org/10.4204/EPTCS.239.2

[5] Bar, K., Kissinger, A., Vicary, J.: Globular: an online proof assistant for higher-dimensional rewriting. Logical Methods in Computer Science **14**(1) (2018), https://doi.org/10.23638/LMCS-14(1:8)2018

[6] Bessai, J., Chen, T.C., Dudenhefner, A., DÃijdder, B., de'Liguoro, U., Rehof, J.: Mixin Composition Synthesis based on Intersection Types. Logical Methods in Computer Science **Volume 14, Issue 1** (Feb 2018). https://doi.org/10.23638/LMCS-14(1:18)2018, https://lmcs.episciences.org/4319

[7] Bessai, J., Düdder, B., Heineman, G.T., Rehof, J.: 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 (2015), https://doi.org/10.1007/978-3-319-28934-2_7

[8] Bessai, J., Düdder, B., Heineman, G.T., et al.: (CL)S Framework (2018), http://www.combinators.org, accessed: 2018-04-30

[9] Bessai, J., Dudenhefner, A., Düdder, B., Martens, M., Rehof, J.: Combinatory Process Synthesis. In: Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. pp. 266–281 (2016), https://doi.org/10.1007/978-3-319-47166-2_19

[10] Blanc, R., Kuncak, V., Kneuss, E., Suter, P.: An overview of the leon verification system: verification by translation to recursive functions. In: Proceedings of the 4th Workshop on Scala. pp. 1:1–1:10 (2013), http://doi.acm.org/10.1145/2489837.2489838

[11] Bobot, F., Filliâtre, J., Marché, C., Paskevich, A.: Let's verify this with Why3. STTT **17**(6), 709–727 (2015), https://doi.org/10.1007/s10009-014-0314-5

[12] Bucciarelli, A., Kesner, D., Rocca, S.R.D.: The Inhabitation Problem for Non-idempotent Intersection Types. In: Theoretical Computer Science. pp. 341–354 (2014), https://doi.org/10.1007/978-3-662-44602-7_26

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

[14] Coppo, M., Dezani-Ciancaglini, M.: A new type assignment for λ-terms. Arch. Math. Log. **19**(1), 139–156 (1978), https://doi.org/10.1007/BF02011875

[15] Düdder, B., Martens, M., Rehof, J., Urzyczyn, P.: Bounded Combinatory Logic. In: Proceedings of the 26th International Workshop on Computer Science Logic. pp. 243–258 (2012), https://doi.org/10.4230/LIPIcs.CSL.2012.243

[16] Düdder, B., Rehof, J., Heineman, G.T.: 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 (2015), http://doi.acm.org/10.1145/2791060.2793677

[17] Dudenhefner, A., Martens, M., Rehof, J.: The Algebraic Intersection Type Unification Problem. Logical Methods in Computer Science **13**(3) (2017), https://doi.org/10.23638/LMCS-13(3:9)2017

[18] Dudenhefner, A., Rehof, J.: Intersection type calculi of bounded dimension. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 653–665 (2017), http://dl.acm.org/citation.cfm?id=3009862

[19] Engelfriet, J., Heyker, L.: Context-Free Hypergraph Grammars have the Same Term-Generating Power as Attribute Grammars. Acta Inf. **29**(2), 161–210 (1992), https://doi.org/10.1007/BF01178504

[20] Felty, A.P., Middeldorp, A. (eds.): Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings, Lecture Notes in Computer Science, vol. 9195. Springer (2015), https://doi.org/10.1007/978-3-319-21401-6

[21] Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex APIs pp. 599–612 (2017), http://dl.acm.org/citation.cfm?id=3009851

[22] Frankle, J., Osera, P., Walker, D., Zdancewic, S.: Example-Directed Synthesis: A Type-Theoretic Interpretation. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 802–815 (2016), http://doi.acm.org/10.1145/2837614.2837629

[23] Franz, M., Lopes, C.T., Huck, G., Dong, Y., Sümer, S.O., Bader, G.D.: Cytoscape.js: a graph theory library for visualisation and analysis. Bioinformatics **32**(2), 309–311 (2016), https://doi.org/10.1093/bioinformatics/btv557

[24] Gécseg, F., Steinby, M.: Tree Automata. CoRR **abs/1509.06233** (2015), http://arxiv.org/abs/1509.06233

[25] Heineman, G.T., Bessai, J., Düdder, B., Rehof, J.: A Long and Winding Road Towards Modular Synthesis. In: Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. pp. 303–317 (2016), https://doi.org/10.1007/978-3-319-47166-2_21

[26] Hillah, L., Kordon, F.: Petri Nets Repository: A Tool to Benchmark and Debug Petri Net Tools. In: Proceedings of the 38th International Conference on Application and Theory of Petri Nets and Concurrency. pp. 125–135 (2017), https://doi.org/10.1007/978-3-319-57861-3_9

[27] Hindley, J.R.: Types with Intersection: An Introduction. Formal Aspects of Computing **4**(5), 470–486 (1992), https://doi.org/10.1007/BF01211394

[28] Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts. Standard, International Organization for Standardization, Geneva, CH (2018)

[29] Kabbani, N.M., Welch, D., Priester, C., Schaub, S., Durkee, B., Sun, Y., Sitaraman, M.: Formal Reasoning Using an Iterative Approach with an Integrated Web IDE. In: Proceedings of the 2nd International Workshop on Formal Integrated Development Environment. pp. 56–71 (2015), https://doi.org/10.4204/EPTCS.187.5

[30] Kaliszyk, C.: Web interfaces for proof assistants. Electr. Notes Theor. Comput. Sci. **174**(2), 49–61 (2007), https://doi.org/10.1016/j.entcs.2006.09.021

[31] Kreowski, H.: A Comparison Between Petri-Nets and Graph Grammars. In: Proceedings of the International Workshop on Graphtheoretic Concepts in Computer Science. pp. 306–317 (1980), https://doi.org/10.1007/3-540-10291-4_22

[32] Lightbend: Play framework, https://www.playframework.com

[33] Myers, B.A., Ko, A.J., LaToza, T.D., Yoon, Y.: Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. IEEE Computer **49**(7), 44–52 (2016), https://doi.org/10.1109/MC.2016.200

[34] Nielsen, J.: Usability Engineering. Elsevier Science (1994)

[35] Vasileva, A., Schmedding, D.: How to Improve Code Quality by Measurement and Refactoring. In: Proceedings of the 10th International Conference on the Quality of Information and Communications Technology. pp. 131–136 (2016), http://doi.ieeecomputersociety.org/10.1109/QUATIC.2016.034

[36] Workshop Series: Intersection Types and Related Systems, http://itrs.di.unito.it/index.html, accessed: 2018-04-25