Staged Composition Synthesis

Boris Düdder, Moritz Martens, and Jakob Rehof

Technical University of Dortmund, Faculty of Computer Science

Abstract. A framework for composition synthesis is provided in which metalanguage combinators are supported and the execution of synthesized programs can be staged into composition-time code generation (stage 1) and run-time execution (stage 2). By extending composition synthesis to encompass both object language (L1) and metalanguage (L2) combinators, composition synthesis becomes a powerful and flexible framework for the generation of L1-program compositions. A system of modal intersection types is introduced into a combinatory composition language to control the distinction between L1- and L2-combinators at the type level, thereby exposing the language distinction to composition synthesis. We provide a theory of correctness of the framework which ensures that generated compositions of component implementations are well typed and that their execution can be staged such that all metalanguage combinators can be computed away completely at stage 1, leaving only well typed L1-code for execution at stage 2. Our framework has been implemented, and we report on experiments.

1 Introduction

Composition synthesis [1–5] is based on the idea of using inhabitation in combinatory logic [6] with intersection types [7] as a foundation for computing compositions from a repository of components. We can regard a combinatory type judgement $\Gamma \vdash e : \tau$ as modeling the fact that combinatory expression e can be obtained by composition from a repository Γ of components which are exposed as combinator symbols and whose interfaces are exposed as combinator types enriched with intersection types that specify semantic properties of components. The decision problem of inhabitation, often indicated as $\Gamma \vdash ? : \tau$, is the question whether a combinatory expression e exists such that $\Gamma \vdash e : \tau$ (such an expression e is called an inhabitant of τ). An algorithm (or semi-algorithm) for solving the inhabitation problem searches for inhabitants and can be used to synthesize them. Under the propositions-as-types correspondence, inhabitation is the question of provability in a Hilbert-style presentation of a propositional logic, where Γ represents a propositional theory, τ represents a proposition to be proved, and e is a proof.

Following [8, 9], a level of *semantic types* is introduced to specify component interfaces and synthesis goals so as to direct synthesis by means of semantic concepts. Semantic types are not necessarily checked against component implementations (this is regarded as an orthogonal issue). In the combinatory approach

Z. Shao (Ed.): ESOP 2014, LNCS 8410, pp. 67–86, 2014.

of [1–5] semantic types are represented by intersection types [7]. In addition to being inherently component-oriented, it is a possible advantage of the type-based approach of composition synthesis that types can be naturally associated with code at the API-level. We think of intersection types as hosting a two-level type system, consisting of *native types* and *semantic types*. Native types are types of the implementation language, whereas semantic types are abstract, application-dependent conceptual structures, drawn, e.g., from a taxonomy of semantic concepts. For example, in the specification

$$X : ((\texttt{real} \times \texttt{real}) \cap \mathit{Cart} \to (\texttt{real} \times \texttt{real}) \cap \mathit{Pol}) \cap \mathit{Isom}$$

native types ($\texttt{real}$, $\texttt{real} \times \texttt{real}, \ldots$) are enriched with semantic types (in the example, Cart, Pol, Isom) by means of intersections. Semantic types express intended properties of the component (combinator) X — e.g., that it is an isometry transforming Cartesian to polar coordinates. We can think of semantic types as organized in any system of finite-dimensional feature spaces (e.g., Cart, Pol are features of coordinates, Isom is a feature of functions) whose elements can be mapped onto the native API using intersections, at any level of the type structure.

In this paper we develop a framework for *staged composition synthesis* (SCS) in which *compositional metalanguage* components, implemented in a distinct language suitable for metaprogramming, can be introduced into composition synthesis. The introduction of metalanguage combinators adds power and flexibility to composition synthesis in several respects, including the ability to define special purpose composition operators, higher-order functional abstraction, native language code template substitution and code-generating operators.

In more detail, we assume here that we have a (possibly low-level) component implementation language $\mathsf{L1}$ in which we can execute programs at runtime, referred to as the *native language*. Following the ideas summarized in [4], components written in $\mathsf{L1}$ can be exposed for composition synthesis through a combinatory environment $\mathscr{C}$ in which named components are exposed as semantically typed combinator symbols $(X : \phi)$, where X is the name of the component and ϕ° is the native type of the component in the language $\mathsf{L1}$ (the map $()^\circ$ erases semantic type information). So we assume for each $(X : \phi) \in \mathscr{C}$ that we have a native implementation program T_X with $\vdash_{\mathsf{L1}} T_X : \phi^\circ$, where $\vdash_{\mathsf{L1}}$ formalizes the type system of $\mathsf{L1}$, and where the implementation T_X is associated with the typed combinator symbol $(X : \phi)$. Composition of $\mathsf{L1}$-components from the environment $\mathscr{C}$ can be formalized in a corresponding combinatory logic, $\mathsf{C1}$. We take a simple monomorphic imperative first-order language as our exemplary native language $\mathsf{L1}$.

In our SCS framework we want to enhance our ability to compute compositions of $\mathsf{L1}$-programs by introducing *templates* of $\mathsf{L1}$-program fragments and expressions into which we can substitute other $\mathsf{L1}$-program expressions to build complex $\mathsf{L1}$-programs from simpler ones. To realize this idea in full, we need a possibly different language, $\mathsf{L2}$, referred to as the *compositional metalanguage*, which is suited for the metaprogramming tasks involved in computing over $\mathsf{L1}$-templates. Since a central task here is to perform substitutions into $\mathsf{L1}$-templates

we take the λ-calculus as our exemplary L2-language. In order to formalize this situation, we introduce type variables (type templates) and special program expression template variables u into L1 to serve as substitutable placeholders for L1-expressions inside other L1-expressions. Moreover, we introduce a type system $\vdash_{L2}$ for the metalanguage L2. Now, if we could compose *both* L1-programs *and* L2-programs that compose L1-programs, we could achieve more flexible and powerful forms of composition, since we can implement special code-generating L1-composition operators in L2, depending on situation and purpose. This situation is formalized by introducing a combinatory logic, C2, in which compositions of L2-programs are computed. In this combinatory logic C2, implemented L2-components are exposed in combinatory environments $\mathscr{D}$ analogously to the way L1-components are exposed in environments $\mathscr{C}$ in C1. We now have two implementation languages, L1 and L2, exposed for combinatory composition through associated combinatory logics, C1 and C2. In this system, we need a phase distinction between *composition time* computations in L2 and *runtime* computations in L1: we first (stage 1) perform composition time computations in the metalanguage L2 which produce L1-programs to be executed at runtime (stage 2). Since our focus is entirely on the generation of L1-compositions, we shall focus on L2-computations here.

Main Technical Contributions. Since composition synthesis is entirely type-directed, we need to expose the language- and phase distinction between L1 and L2 to synthesis at the type level. We solve this problem by exploiting the ideas of *staged computation* introduced by Davies and Pfenning [10], using modal types of the form $\Box\tau$ to describe "code of type τ". In our setting, such a type can appear in an L2-program manipulating L1-code to describe L1-code with L1-type τ. The type system ensures that L2-computations over L1-code is sound, i.e., that L2-implementations of type $\Box\tau$ can be computed away completely at composition time in L2 leaving a well typed L1-program (of type τ) as a result.

Our main technical innovation is the design and theory of semantic types at the combinatory logic level (C1 and C2), which are based on a novel system of *modal intersection types*. Such types can be superimposed onto implementation language types (L1 and L2) to express semantic properties of components to control composition. The basic challenge here is to achieve a design which allows such highly expressive semantic types to coexist with a guarantee of *implementation type correctness* (cf. Thm. 1), i.e., that synthesized compositions remain well typed in the implementation languages under semantic type erasure, even though compositions are constructed in a much more expressive type system of intersection types.

Our framework has been implemented in an extension of the (CL)S (Combinatory Logic Synthesizer) tool, and we report on the results of experiments using the tool for SCS.

Organization of the Paper. The remainder of this paper is organized as follows. In Sec. 2 we introduce the native language L1 and the metalanguage L2 (some definitions are placed in App.(s) A and B at the end of the paper).

Semantic types are defined in Sec. 3, and the combinatory logics C1 and C2 are defined in Sec. 4. In Sec. 5 we consider a simple example to illustrate SCS. In Sec. 6 we develop the theory of implementation type correctness, and Sec. 7 is devoted to the inhabitation algorithm underlying our extension of (CL)S, and experiments with the tool are discussed in Sec. 8. Related work is discussed in Sec. 9, and Sec. 10 concludes the paper.

2 Implementation Languages

We introduce an exemplary native language L1 and a compositional metalanguage L2, referred to collectively as implementation languages. In distinction to the framework of Davies and Pfenning [10] we have two distinct languages, which are highly independent of each other (regarding both operational semantics and type systems). Moreover, we only wish to distinguish exactly two stages of computation, *runtime* computation in the native language and *composition time* computation in the metalanguage (in [10] arbitrary levels of stages exist within a single language, and our framework can also be thus generalized). Our goal is a framework in which the native language is largely substitutable – native programs are regarded as "black boxes" that are exposed as expressions box T to the language L2 with L2-types of the form $\Box\tau$ (where τ is an L1-type), but other than that the theory of L2 is agnostic of the nature of programs T and types τ of L1.

For concreteness, we fix a simply typed first-order core language as an exemplary native implementation language L1 shown in Fig. 5, App. A, but (as mentioned) L1 can be exchanged easily. The only requirements on the design of L1 are that L1 should be typed, it should contain functions and function application, the language should satisfy preservation of types under appropriate term substitution (see substitution Lem. 1, App. A, for L1), and that well typed L1-programs can be executed at a later runtime stage (with which we are not further concerned, here). The native language consists of template expressions T containing template variables ranged over by u. Other native expressions or templates can be substituted for template variables. The type structure consists of a set $\mathbb{T}_0$ of *value types* ranged over by t_0, reference types t_1 and the set of *native template types* $\mathbb{T}_1$, ranged over by τ in which value types can be substituted for type variables ranged over by $\alpha, \beta, \gamma, \ldots \in \mathbb{V}$ (cf. App. A). We do not specify an operational semantics for L1, since it is altogether standard, and we are mainly concerned with computations in the metalanguage which we will consider next.

The compositional metalanguage L2 is a standard λ-calculus with simple types extended with *modal types* as introduced by Davies and Pfenning [10] to distinguish computational stages at the type level. In our setting, we can intuitively understand an L2-type $\Box\tau$ ($\tau \in \mathbb{T}_1$) as meaning "L1-code with L1-type τ". The set $\mathbb{T}_2$ denotes *metalanguage types*, ranged over by σ. The modal type constructor $\Box$ is a special covariant constructor.

$$\mathbb{T}_2 \ni \sigma ::= \Box\tau \mid \sigma \to \sigma'$$

Compositional metalanguage terms are terms M of the $\lambda_e^{\Box\to}$-calculus [10]:

$$M ::= \mathsf{box}\ T \mid \mathsf{letbox}\ u : \tau = M_1\ \mathsf{in}\ M_2 \mid x \mid \lambda x : \sigma.M \mid (M_1 M_2)$$

Compositional metalanguage expressions are typed by the system L2 shown in Fig. 1. Judgements are of the form $\Delta; \Gamma \vdash_{\mathsf{L2}} M : \sigma$, where Δ contains L1-bindings $(u : \tau)$ of native template variables to L1-types, and Γ is the standard λ-calculus type environment of bindings $(x : \sigma)$ of λ-variables to L2-types.

The rule ($\Box$I) together with the environment Δ provide the interface between L1 and L2. According to this rule, native templates T that are well typed with native template types in L1 can be injected into L2 by being placed in the scope of the box-operator. Importantly, the rule requires that we only inject native *expressions* T with no free native program variables (but possibly with free template variables) into L2. As shown in [10], this discipline ensures that we can soundly substitute native expressions into native templates in L2-computations. The dual rule ($\Box$E) discharges assumptions in Δ using the letbox construct. As detailed in App. B, this construct performs substitution of native templates into native template variables under L2-computation.

$$\frac{}{\Delta; (\Gamma, x : \sigma) \vdash_{\mathsf{L2}} x : \sigma}(\mathrm{var})$$

$$\frac{\Delta; (\Gamma, x : \sigma) \vdash_{\mathsf{L2}} M : \sigma'}{\Delta; \Gamma \vdash_{\mathsf{L2}} \lambda x : \sigma.M : \sigma \to \sigma'}(\to\mathrm{I}) \qquad \frac{\Delta; \Gamma \vdash_{\mathsf{L2}} M_1 : \sigma \to \sigma' \quad \Delta; \Gamma \vdash_{\mathsf{L2}} M_2 : \sigma}{\Delta; \Gamma \vdash_{\mathsf{L2}} (M_1 M_2) : \sigma'}(\to\mathrm{E})$$

$$\frac{\Delta; \emptyset \vdash_{\mathsf{L1}} T : \tau}{\Delta; \Gamma \vdash_{\mathsf{L2}} \mathsf{box}\ T : \Box\tau}(\Box\mathrm{I}) \qquad \frac{\Delta; \Gamma \vdash_{\mathsf{L2}} M_1 : \Box\tau \quad (\Delta, u : \tau); \Gamma \vdash_{\mathsf{L2}} M_2 : \sigma}{\Delta; \Gamma \vdash_{\mathsf{L2}} \mathsf{letbox}\ u : \tau = M_1\ \mathsf{in}\ M_2 : \sigma}(\Box\mathrm{E})$$

Fig. 1. Metalanguage L2

The operational semantics of L2 are summarized in App. B. As a consequence of theorems in [10] (subject reduction, Thm. 4, and eliminability, Thm. 5, see App. B), typability in system L2 implies that reducing, in L2, an expression of type $\Box\tau$ to normal form results in a well typed native L1-program in the scope of a box-constructor. In sum, it is guaranteed for a well typed closed L2-term of type $\Box\tau$ that *composition-time* reduction to normal form in L2 computes all L2-term occurrences away and leaves only a well typed boxed L1-program as a result. That L1-term can then be executed at the next stage (*run-time*).

3 Semantic Types

We introduce a level of semantic types, which are special structures of modal intersection types, to be used in the combinatory logics C1 and C2 (Sec. 4).

The sets $\mathfrak{S}_i$ of *semantic types* of level i ($i = 1, 2$) are ranged over by $\mathfrak{t}$ and $\mathfrak{s}$, respectively. Type variables of level 1 are ranged over by $\mathfrak{a} \in \mathbb{V}_{\mathfrak{S}_1}$, and type variables of level 2 are ranged over by $\mathfrak{b} \in \mathbb{V}_{\mathfrak{S}_2}$. We assume that $\mathbb{V}$, $\mathbb{V}_{\mathfrak{S}_1}$ and $\mathbb{V}_{\mathfrak{S}_2}$ are disjoint sets. We assume sets of semantic type constants $\mathbb{D}_1$ and $\mathbb{D}_2$, with $\mathbb{D}_1$ and $\mathbb{D}_2$ disjoint from each other and from the constants of $\mathsf{L1}$.

The set $\mathfrak{S}_1$ contains a copy[1] of $\mathbb{T}_1$ built from distinct sets of variables $\mathbb{V}_{\mathfrak{S}_1}$ and constants $\mathbb{D}_1$ and closed under intersection. The set $\mathfrak{S}_2$ is built analogously, as a copy of $\mathbb{T}_2$ over distinct variables in $\mathbb{V}_{\mathfrak{S}_2}$ and constants in $\mathbb{D}_2$:

$$\mathfrak{S}_1 \ni \mathfrak{t} ::= \mathfrak{a} \mid d_1 \mid \mathfrak{t} \to \mathfrak{t}' \mid \mathfrak{t} \cap \mathfrak{t}' \qquad \mathfrak{S}_2 \ni \mathfrak{s} ::= \mathfrak{b} \mid d_2 \mid \mathfrak{s} \to \mathfrak{s}' \mid \mathfrak{s} \cap \mathfrak{s}' \mid \Box\mathfrak{t}$$

The set of *semantic* $\mathsf{L1}$*-types* $\mathbb{S}_1$ is ranged over by ϕ, and the set of *semantic* $\mathsf{L2}$*-types* $\mathbb{S}_2$ is ranged over by ψ:

$$\mathbb{S}_1 \ni \phi ::= \tau \mid \phi \cap \mathfrak{t} \mid \mathfrak{t} \cap \phi \mid \phi \to \phi' \mid \phi \cap \phi' \qquad \mathbb{S}_2 \ni \psi ::= \sigma \mid \psi \cap \mathfrak{s} \mid \mathfrak{s} \cap \psi \mid \Box\phi \mid \psi \to \psi' \mid \psi \cap \psi'$$

We follow the convention that $\to$ is right-associative and that $\cap$ binds stronger than $\to$. Type expressions are implicitly considered as equivalence classes modulo commutativity, associativity and idempotency of $\cap$. We let $\vartheta, \varrho, \upsilon$ range over $\mathbb{S}_1 \cup \mathbb{S}_2 \cup \mathfrak{S}_1 \cup \mathfrak{S}_2$. Notice that $\mathbb{T}_1$ and $\mathbb{T}_2$ are disjoint, $\mathbb{S}_1$ and $\mathbb{S}_2$ are disjoint, and $\mathbb{T}_i \subseteq \mathbb{S}_i$. An *atom* is a type variable or a type constant, and we let A range over atoms.

The semantic type structure allows for maximal freedom in combining semantic types with "underlying" (see Def. 3, Sec. 6) implementation types and allows us to treat the semantic types as a distinct kind from implementation types. The type structures $\mathbb{T}_0$ and $\mathfrak{S}_i$ are treated as different kinds in the following definition of type substitution, which ensures that the sets $\mathbb{T}_i$, $\mathbb{S}_i$, and $\mathfrak{S}_i$ ($i = 1, 2$) are closed under substitutions.

Definition 1. *A* type substitution *is a map* $S : \mathbb{V} \cup \mathbb{V}_{\mathfrak{S}_1} \cup \mathbb{V}_{\mathfrak{S}_2} \to \mathbb{T}_0 \cup \mathfrak{S}_1 \cup \mathfrak{S}_2$ *satisfying the following conditions:* $\forall \alpha \in \mathbb{V}.\ S(\alpha) \in \mathbb{T}_0$, $\forall \mathfrak{a} \in \mathbb{V}_{\mathfrak{S}_1}.\ S(\mathfrak{a}) \in \mathfrak{S}_1$, *and* $\forall \mathfrak{b} \in \mathbb{V}_{\mathfrak{S}_2}.\ S(\mathfrak{b}) \in \mathfrak{S}_2$.

The following definition is standard for intersection types [7]. We tacitly assume that the sets $\mathbb{D}_1$ and $\mathbb{D}_2$ can be equipped with partial orders $\leq_{\mathbb{D}_i}$ in which case axioms $d_i \leq_{\mathbb{D}_i} d_i' \Rightarrow d_i \leq d_i'$ are added to the axiomatization of subtyping.

Definition 2. Subtyping $\leq$ *is the least preorder (reflexive and transitive relation) on* $\mathbb{S}_1 \cup \mathbb{S}_2 \cup \mathfrak{S}_1 \cup \mathfrak{S}_2$*, satisfying the following conditions:*

$$\vartheta \cap \varrho \leq \vartheta, \quad \vartheta \cap \upsilon \leq \upsilon, \quad (\vartheta \to \varrho) \cap (\vartheta \to \upsilon) \leq \vartheta \to \varrho \cap \upsilon,$$
$$\vartheta \leq \vartheta' \wedge \varrho \leq \varrho' \Rightarrow \vartheta' \to \varrho \leq \vartheta \to \varrho', \quad \vartheta \leq \vartheta' \wedge \varrho \leq \varrho' \Rightarrow \vartheta \cap \varrho \leq \vartheta' \cap \varrho',$$
$$(\Box\vartheta) \cap (\Box\varrho) \leq \Box(\vartheta \cap \varrho), \quad \vartheta \leq \varrho \Rightarrow \Box\vartheta \leq \Box\varrho.$$

We say that ϑ *and* ϱ *are* equal, *written* $\vartheta = \varrho$*, if* $\vartheta \leq \varrho$ *and* $\varrho \leq \vartheta$*. We write* $\vartheta \equiv \varrho$*, if* ϑ *and* ϱ *are syntactically identical.*

[1] Function types of $\mathfrak{S}_1$ and $\mathbb{S}_1$ below are not restricted to be first order, since our system is developed for the general case. If $\mathsf{L1}$ happens to be restricted to a first-order system, as in our example case, our semantic type framework is more general.

The following distributivity properties follow from the axioms of subtyping:

$$(\vartheta \to \varrho) \cap (\vartheta \to \upsilon) = \vartheta \to \varrho \cap \upsilon, \quad (\vartheta \to \varrho) \cap (\vartheta' \to \varrho') \leq (\vartheta \cap \vartheta') \to (\varrho \cap \varrho'),$$
$$(\Box\vartheta) \cap (\Box\varrho) = \Box(\vartheta \cap \varrho).$$

4 Combinatory Logic and Composition Synthesis

We introduce combinatory logics C1 and C2 in which components implemented in L1 and L2 can be exposed as combinator symbols. The combinatory rules of C1 are standard for combinatory logic with intersection types [11], and the rules of C2 are extended (in rule ($\Box$I)) according to the modal extension of L2. Combinatory C1-terms are defined by $e ::= X \mid (e_1 e_2)$. Environments $\mathscr{C}$ are finite sets of bindings of the form $(X : \phi)$ with $\phi \in \mathbb{S}_1$. The combinatory logic C1 is defined by the rules of Fig. 2. Combinatory C2-terms are defined by $E ::= F \mid (E_1 E_2) \mid \mathsf{box}\ e$. Environments $\mathscr{D}$ are finite sets of bindings of the form $(F : \psi)$ with $\psi \in \mathbb{S}_2$. The combinatory logic C2 is defined by the rules of Fig. 3. Note carefully, that the rules ($\leq$) for C1 (resp. C2) are restricted to types in $\mathbb{S}_1$ (resp. $\mathbb{S}_2$) as shown by use of the metavariable ϕ (resp. ψ). This restriction is necessary for Thm. 1 to go through.

$$\frac{}{\mathscr{C}, X : \phi \vdash_{\mathsf{C1}} X : S(\phi)}(\text{var}) \qquad \frac{\mathscr{C} \vdash_{\mathsf{C1}} e_1 : \phi \to \phi' \quad \mathscr{C} \vdash_{\mathsf{C1}} e_2 : \phi}{\mathscr{C} \vdash_{\mathsf{C1}} (e_1 e_2) : \phi'}(\to\text{E})$$

$$\frac{\mathscr{C} \vdash_{\mathsf{C1}} e : \phi \quad \mathscr{C} \vdash_{\mathsf{C1}} e : \phi'}{\mathscr{C} \vdash_{\mathsf{C1}} e : \phi \cap \phi'}(\cap\text{I}) \qquad \frac{\mathscr{C} \vdash_{\mathsf{C1}} e : \phi \quad \phi \leq \phi'}{\mathscr{C} \vdash_{\mathsf{C1}} e : \phi'}(\leq)$$

Fig. 2. Combinatory logic C1

$$\frac{}{\mathscr{C}; (\mathscr{D}, F : \psi) \vdash_{\mathsf{C2}} F : S(\psi)}(\text{var})$$

$$\frac{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E_1 : \psi \to \psi' \quad \mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E_2 : \psi}{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} (E_1 E_2) : \psi'}(\to\text{E}) \qquad \frac{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E : \psi \quad \mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E : \psi'}{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E : \psi \cap \psi'}(\cap\text{I})$$

$$\frac{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E : \psi \quad \psi \leq \psi'}{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E : \psi'}(\leq) \qquad \frac{\mathscr{C} \vdash_{\mathsf{C1}} e : \phi}{\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} \mathsf{box}\ e : \Box\phi}(\Box\text{I})$$

Fig. 3. Combinatory logic C2

In composition synthesis (see [4] for a general introduction), we are concerned with the *relativized inhabitation problem*: Given $\mathscr{C}$, $\mathscr{D}$ and ψ, does there exist a combinatory term E such that $\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} E : \psi$? We use the notation $\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} ? : \psi$

to specify the problem. The inhabitation relation in combinatory logic can be used as a foundation for component-oriented synthesis, since an inhabitation algorithm can be employed to compute program terms E (inhabitants) by combinatory composition. Here we think of $\mathscr{C}$ and $\mathscr{D}$ as component repositories and ψ as a synthesis goal specification.

It should be noted that the expressive power of the inhabitation relation is enormous: it is undecidable even in simple types. The reason is that the relation is not confined to a fixed base but is *relativized* to arbitrary environments $\mathscr{C}$, $\mathscr{D}$ ([4] contains a survey of the relevant results). We can indeed consider the inhabitation relation as an operational semantics for a Turing-complete abstract logic programming language [12] at the level of interface types, as suggested in [4]. Or, we can restrict the relation to ensure that search for inhabitants always terminates. We consider algorithmic aspects of the relation in Sec. 7.

We emphasize again that our design of semantic types must be understood in conjunction with the theory of implementation type correctness given by Thm. 1 (Sec. 6), which guarantees preservation of typability of compositions in the implementation languages under semantic type erasure (Def. 3). The semantic type structure in Sec. 3 is a restricted yet very general structure still allowing Thm. 1 to go through. The fundamental challenge here is to ensure that implementation language typability is guaranteed even though compositions are constructed in a much more expressive intersection type theory. Freely combining semantic types with implementation types under intersection and subtyping will not work — as a simple example, one would derive $\{X : \mathtt{bool} \cap (d \to d'), Y : \mathtt{int} \cap d\} \vdash (XY) : d'$ (where d, d' are semantic type constants in $\mathfrak{S}_1$ and $\mathtt{bool}, \mathtt{int}$ are $\mathsf{L1}$-types) which does not type check when semantic types are erased. Our system prevents such problems by ensuring that semantic types have sufficient support (see Def. 4) in the implementation language (in the example above, using rule $\leq$ to derive $X : d \to d'$ is not allowed even though $\mathtt{bool} \cap (d \to d') \leq d \to d'$, since $d \to d'$ is not a member of $\mathbb{S}_1$).

5 Example

We introduce a simple example adapted from [4] and extended with our modal types to illustrate a few basic features of the formal system. For ease of reading, we write $\mathfrak{S}_1$-types in *blue* font (as in *Cel*) and $\mathfrak{S}_2$-types in *red* font (as in *Conv*). Programs and combinators in $\mathsf{L1}$ are written in `green` typewriter font. Whenever convenient we use the shorthand notation $\tilde{\tau}$ to denote an $\mathbb{S}_1$-type consisting of the $\mathsf{L1}$-type τ intersected with an associated semantic type variable $\mathfrak{a}_\tau$ from $\mathfrak{S}_1$, so, for example, $\tilde{\alpha}$ denotes the type $\alpha \cap \mathfrak{a}_\alpha$. $\mathsf{L1}$-types are written in black typewriter font, for example, $\mathtt{R}$ which denotes the type of reals.

Let $\mathscr{C}$ contain the combinators (we freely extend the $\mathsf{L1}$-type language by type constructors — below, $(\cdot,\cdot)$ is a pair-constructor whereas $D((\cdot,\cdot),\cdot,\cdot)$ is a

constructor for a data-structure):

$$\begin{array}{l}\texttt{O} \quad : \texttt{TrObj}\\ \texttt{Tr} \quad : \texttt{TrObj} \to \texttt{D}((\texttt{R},\texttt{R}) \cap \mathit{Cart}, \texttt{R} \cap \mathit{Gpst}, \texttt{R} \cap \mathit{Cel})\\ \texttt{tmp} : \texttt{D}((\texttt{R},\texttt{R}), \texttt{R}, \texttt{R} \cap \mathfrak{a}) \to \texttt{R} \cap \mathfrak{a} \cap \mathit{ms}\end{array}$$

The environment $\mathscr{C}$ could be part of a semantic repository of components to track (`Tr`) an object (`O`) by giving the Cartesian coordinates (*Cart*) of the tracked object (`TrObj`) at a given point in time (*Gpst*) and its temperature (*Cel*). There is also a function `tmp` which projects the temperature. Its result type has the semantic component *ms* which is intended to indicate that the datum (in this case, a real number) represents a measurement. Let $\mathscr{D}$ contain the combinators

$$\begin{array}{ll}\bullet & : \Box(\tilde{\beta} \to \tilde{\gamma}) \to \Box(\tilde{\alpha} \to \tilde{\beta}) \to \Box(\tilde{\alpha} \to \tilde{\gamma})\\ \texttt{cl2fh} & : (\Box(\texttt{R} \cap \mathit{Cel}) \to \Box(\texttt{R} \cap \mathit{Fh})) \cap \mathit{Conv}\\ \Diamond & : \Box(\tilde{\alpha} \cap \mathit{ms}) \to (\Box\tilde{\alpha} \to \Box\tilde{\beta}) \cap \mathit{Conv} \to \Box(\tilde{\beta} \cap \mathit{ms})\end{array}$$

with the following bindings to implementations in L2:

$$\begin{array}{ll}\bullet & \triangleq \lambda G : \Box(\beta \to \gamma).\lambda F : \Box(\alpha \to \beta).\\ & \quad \mathsf{letbox}\ f : \alpha \to \beta = F\ \mathsf{in}\\ & \quad\quad \mathsf{letbox}\ g : \beta \to \gamma = G\ \mathsf{in}\ \mathsf{box}\ (\texttt{fn y} : \alpha \texttt{ => } (g\ (f\ \texttt{y})))\\ \texttt{cl2fh} & \triangleq \lambda z : \Box\texttt{R}.\\ & \quad \mathsf{letbox}\ u : \texttt{R} = z\ \mathsf{in}\\ & \quad\quad \mathsf{box}\ \texttt{let x : R} = u\ \texttt{in x} * (\texttt{9 div 5}) + \texttt{32}\\ \Diamond & \triangleq \lambda z : \Box\alpha.\lambda F : \Box\alpha \to \Box\beta.(F\ z)\end{array}$$

The semantic $\mathfrak{S}_2$-type *Conv* of the combinator `cl2fh` expresses the idea that the corresponding function acts as a unit conversion. The type of the combinator $\Diamond$ uses this type and the $\mathfrak{S}_1$-type *ms* to express the idea that a conversion can be applied to a measurement to produce something which is still a measurement.

Suppose we ask for a function composable from the component repositories $\mathscr{C}$ and $\mathscr{D}$ which measures the temperature of an object in Celsius. We can formalize this query as the inhabitation problem $\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}}? : \Box(\texttt{TrObj} \to (\texttt{R} \cap \mathit{Cel} \cap \mathit{ms}))$ which has the solution $(\mathsf{box}\ \texttt{tmp}) \bullet (\mathsf{box}\ \texttt{Tr}) : \Box(\texttt{TrObj} \to (\texttt{R} \cap \mathit{Cel} \cap \mathit{ms}))$ where we write $\bullet$ in infix notation. Performing the L2-reduction $(\mathsf{box}\ \texttt{tmp}) \bullet (\mathsf{box}\ \texttt{Tr}) \longmapsto^* \mathsf{box}\ (\texttt{fn y : TrObj => (tmp (Tr y))})$ we see that L1-code implementing such a function is produced.

If we ask for $\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}}? : \Box(\texttt{R} \cap \mathit{Fh} \cap \mathit{ms})$, the solution is, again writing $\Diamond$ in infix notation, $(\mathsf{box}\ (\texttt{tmp (Tr O)}))\ \Diamond\ \texttt{cl2fh}$ with the L2-reduction

$$(\mathsf{box}\ (\texttt{tmp (Tr O)}))\Diamond\texttt{cl2fh} \mapsto^* \mathsf{box}\ \texttt{let x : R} = \texttt{tmp (Tr O) in x}*(\texttt{9 div 5}) + \texttt{32}$$

If we add `c2f` : $(\texttt{R} \cap \mathit{Cel} \cap \mathit{ms}) \to (\texttt{R} \cap \mathit{Fh} \cap \mathit{ms})$ to $\mathscr{C}$, we get the additional solution, $\mathsf{box}\ (\texttt{c2f (tmp (Tr O))})$, for the same inhabitation goal. If we further add the modal apply combinator `mapply` : $\Box(\tilde{\alpha} \to \tilde{\beta}) \to \Box\tilde{\alpha} \to \Box\tilde{\beta}$ with definition

$$\begin{array}{l}\texttt{mapply} \triangleq \lambda F : \Box(\alpha \to \beta).\lambda z : \Box\alpha.\\ \quad\quad \mathsf{letbox}\ f : \alpha \to \beta\ = F\ \mathsf{in}\\ \quad\quad\quad \mathsf{letbox}\ u : \alpha = z\ \mathsf{in}\ \mathsf{box}\ (f\ u)\end{array}$$

to $\mathscr{D}$ we also get `mapply` (box `c2f`) (`mapply` ((box `tmp`) • (box `Tr`))(box `0`)) reducing in L2 to the L1-program box (`c2f` ((`fn y : TrObj => (tmp (Tr y))) 0`)). These examples illustrate that the inhabitation relation determines the possible placements of the box-constructor, in each case determining a specific "division of labor" between L1 and L2. Furthermore, higher-order abstraction in L2 adds considerable power to composition and generation of first-order L1-code.

All examples have been automated by an inhabitation algorithm in our combinatory logic synthesis framework (CL)S as discussed in Sec.(s) 7 and 8.

6 Implementation Type Correctness

Correctness of our framework is based on a conservative extension theorem (Thm. 1) showing that combinatory compositions performed over restricted (supported or grounded, Def. 4) environments can be transformed into well typed L1-expressions. The proof of Thm. 1 depends on a series of technical lemmas which can be found in [13]. Here, we only present the necessary definitions and the theorem. We explain how it constitutes a theory of implementation type correctness for SCS. The following notion relates semantic types to implementation types.

Definition 3 (Erasure). *For type expressions $\vartheta \in \mathbb{S}_i$ we define the* erasure *of ϑ, written ϑ°, $(i = 1, 2)$, as follows.*

$$\begin{array}{ll} \vartheta^\circ & \equiv \vartheta, \ \textit{when } \vartheta \in \mathbb{T}_1 \cup \mathbb{T}_2 \\ (\vartheta \cap \mathfrak{u})^\circ & \equiv \vartheta^\circ \ \textit{when } \mathfrak{u} \in \mathfrak{S}_1 \cup \mathfrak{S}_2 \\ (\mathfrak{u} \cap \vartheta)^\circ & \equiv \vartheta^\circ \ \textit{when } \mathfrak{u} \in \mathfrak{S}_1 \cup \mathfrak{S}_2 \\ (\vartheta \to \varrho)^\circ & \equiv \vartheta^\circ \to \varrho^\circ \\ (\vartheta \cap \varrho)^\circ & \equiv \vartheta^\circ \cap \varrho^\circ \\ (\Box\vartheta)^\circ & \equiv \Box\vartheta^\circ \end{array}$$

For combinatory environments $\mathscr{C}, \mathscr{D}$ we lift $()^\circ$ by pointwise application to the types in the environment, $\mathscr{C}^\circ = \{(X : \phi^\circ) \mid (X : \phi) \in \mathscr{C}\}$ and similarly for $\mathscr{D}^\circ$.

It can be shown that $\vartheta_1 = \vartheta_2$ implies ${\vartheta_1}^\circ = {\vartheta_2}^\circ$, hence the operation $()^\circ$ is a well defined function on equivalence classes with respect to $=$. The function $()^\circ$ erases semantic types in $\mathfrak{S}_1 \cup \mathfrak{S}_2$ from semantic L1- and L2-types in $\mathbb{S}_1 \cup \mathbb{S}_2$.

We introduce combinatory expressions over combinator symbols of C1 and C2, subscripted with types from $\mathbb{T}_1$ and $\mathbb{T}_2$, respectively, as follows.

$$f ::= X_\tau \mid (f_1 f_2) \qquad\qquad g ::= F_\sigma \mid (g_1 g_2) \mid \mathsf{box}\ f$$

For a C1-environment $\mathscr{C}$ in which all bindings are of the form $(X : \bigcap_{j\in J} \tau_j)$ (intersections of types in $\mathbb{T}_1$) and a C2-environment $\mathscr{D}$ in which all bindings are of the form $(F : \bigcap_{j\in J} \sigma_j)$ (intersections of types in $\mathbb{T}_2$) we define the $\mathbb{T}_1$-environment $\mathscr{C}^+$ and the $\mathbb{T}_2$-environment $\mathscr{D}^+$ by

$$\begin{array}{l} \mathscr{C}^+ = \{(X_{\tau_j} : \tau_j) \mid j \in J, (X : \bigcap_{j\in J} \tau_j) \in \mathscr{C}\} \\ \mathscr{D}^+ = \{(F_{\sigma_j} : \sigma_j) \mid j \in J, (F : \bigcap_{j\in J} \sigma_j) \in \mathscr{D}\} \end{array}$$

We can consider such environments $\mathscr{C}^+$ and $\mathscr{D}^+$ as L1-environments, resp. L2-environments, by considering (possibly through a mapping, which we shall leave implicit) the symbols X_τ (resp. F_σ) as L1-variables (resp. L2-variables).

We define an erasure function $()^-$ mapping these expressions back to combinator expressions of C1, respectively C2:

$$\begin{array}{lll} X_\tau^- & \equiv X & F_\sigma^- \equiv F \\ (f_1 f_2)^- & \equiv (f_1^- f_2^-) & (g_1 g_2)^- \equiv (g_1^- g_2^-) \\ & & (\mathsf{box}\ f)^- \equiv \mathsf{box}\ f^- \end{array}$$

The following definition is central for Thm. 1. The concepts of supported and grounded types capture relations between semantic types and their "underlying" (under erasure) implementation language types.

Definition 4 (Supported, grounded). *For $\vartheta \in \mathbb{S}_i$ $(i = 1, 2)$ we say that ϑ is* supported *if $\vartheta = \bigcap_{j \in J} \vartheta_j$ with $\vartheta_j \in \mathbb{T}_i$ for $j \in J$. We say that ϑ is* grounded *if $\vartheta^\circ \in \mathbb{T}_i$. An environment $\mathscr{C}$ or $\mathscr{D}$ is said to be supported (grounded) if all types appearing in the environment are supported (grounded).*

We say that a derivation in C1 or C2 is *monomorphic*, if all applications of rule (var) use the identity substitution (combinatory logics restricted to such derivations are finite combinatory logics, in the sense of [1]). For a derivation tree D in C1 or C2, let $\mathcal{S}_D(X)$ (resp. $\mathcal{S}_D(F)$) be the set of substitutions S such that S is applied in an application in D of rule (var) of the form $\mathscr{C}', X : \phi \vdash_{\mathsf{C1}} X : S(\phi)$ (resp. $\mathscr{C}; (\mathscr{D}', F : \psi) \vdash_{\mathsf{C2}} F : S(\psi)$). We let $\mathscr{C}^D$, resp. $\mathscr{D}^D$, denote the *exponentiated* environments defined as follows:

$$\begin{array}{l} \mathscr{C}^D = \{(X : \bigcap_{S \in \mathcal{S}_D(X)} S(\phi) \mid (X : \phi) \in \mathscr{C}\} \\ \mathscr{D}^D = \{(F : \bigcap_{S \in \mathcal{S}_D(F)} S(\psi) \mid (F : \psi) \in \mathscr{D}\} \end{array}$$

We write $\mathscr{C} \vdash^D_{\mathsf{C1}} e : \phi$ whenever $\mathscr{C} \vdash_{\mathsf{C1}} e : \phi$ is derivable by derivation D, and similarly for $\mathscr{C}; \mathscr{D} \vdash^D_{\mathsf{C2}} E : \psi$. It can be shown that exponentiation preserves supportedness of environments. We write $()^{\circ D} = ((){}^\circ)^D$ and $()^{\circ D+} = (((){}^\circ)^D)^+$. We can now state the conservative extension theorem.

Theorem 1 (Conservative extension).

1. *Suppose that $\mathscr{C}^\circ$ and $\mathscr{D}^\circ$ are supported and that ϕ and ψ are grounded. Then*
   (a) *If $\mathscr{C} \vdash^D_{\mathsf{C1}} e : \phi$ then $\emptyset; \mathscr{C}^{\circ D+} \vdash_{\mathsf{L1}} f : \phi^\circ$ for some f with $f^- \equiv e$.*
   (b) *If $\mathscr{C}; \mathscr{D} \vdash^D_{\mathsf{C2}} E : \psi$ then $\emptyset; \mathscr{C}^{\circ D+}, \mathscr{D}^{\circ D+} \vdash_{\mathsf{L2}} g : \psi^\circ$ for some g with $g^- \equiv E$.*
2. *Suppose that $\mathscr{C}, \mathscr{D}, \phi$, and ψ are grounded. Then*
   (a) *If $\mathscr{C} \vdash^D_{\mathsf{C1}} e : \phi$ with D monomorphic then $\emptyset; \mathscr{C}^\circ \vdash_{\mathsf{L1}} e : \phi^\circ$.*
   (b) *If $\mathscr{C}; \mathscr{D} \vdash^D_{\mathsf{C2}} E : \psi$ with D monomorphic then $\emptyset; \mathscr{C}^\circ, \mathscr{D}^\circ \vdash_{\mathsf{L2}} E : \psi^\circ$.*

We use the theorem to show that inhabitants obtained by composition in C1 and C2 from environments whose erasure represent well typed programs in L1 and L2 can be translated back to well typed expressions of L1 and L2 by type instantiations.

Assume that we have sets of combinator symbols $\mathcal{X}$ and $\mathcal{F}$, and assume that $X \in \mathcal{X}$ and $F \in \mathcal{F}$ are associated with implementations T_X and M_F in $\mathsf{L1}$ and $\mathsf{L2}$, respectively, such that

$$\emptyset;\emptyset \vdash_{\mathsf{L1}} T_X : \tau_X \text{ for } X \in \mathcal{X} \qquad \emptyset;\emptyset \vdash_{\mathsf{L2}} M_F : \sigma_F \text{ for } F \in \mathcal{F}$$

and exposed in combinatory environments as

$$\mathscr{C} = \{(X : \phi_X) \mid X \in \mathcal{X}\} \qquad \mathscr{D} = \{(F : \psi_F) \mid F \in \mathcal{F}\}$$

with $(\phi_X)^\circ = \tau_X$ and $(\psi_F)^\circ = \sigma_F$. Then $\mathscr{C}$ and $\mathscr{D}$ are grounded, hence $\mathscr{C}^\circ$ and $\mathscr{D}^\circ$ are supported and $\mathscr{C}^{\circ D+}$ and $\mathscr{D}^{\circ D+}$ are grounded.

Suppose now that $\mathscr{C} \vdash^D_{\mathsf{C1}} e : \phi$ with ϕ grounded. It follows from Thm. 1 that we have $\emptyset;\mathscr{C}^{\circ D+} \vdash_{\mathsf{L1}} f : \phi^\circ$ for some f with $f^- \equiv e$. Since all combinators in $\mathscr{C}^{\circ D+}$ have the form $(X_{S(\tau_X)} : S(\tau_X))$ with $(X : \phi_X) \in \mathscr{C}$ and $(\phi_X)^\circ \equiv \tau_X$, it follows by Lem. 1, App. A that we have $\emptyset;\emptyset \vdash_{\mathsf{L1}} f' : \phi^\circ$ where

$$f' \equiv f[X_{S(\tau_X)} := S(T_X)]$$

Similarly, if $\mathscr{C};\mathscr{D} \vdash_{\mathsf{C2}} E : \psi$ with ψ grounded, we have $\emptyset;\mathscr{C}^{\circ D+}, \mathscr{D}^{\circ D+} \vdash_{\mathsf{L2}} g : \phi^\circ$ with bindings in $\mathscr{D}^{\circ D+}$ all of the form $(F_{S(\sigma_F)} : S(\sigma_F))$ with $(F : \psi_F) \in \mathscr{D}$ and $(\psi_F)^\circ \equiv \sigma_F$. Hence, by Lem. 2, App. B we have $\emptyset;\emptyset \vdash_{\mathsf{L2}} g' : \psi^\circ$ where

$$g' \equiv g[X_{S(\tau_X)} := S(T_X)][F_{S(\sigma_F)} := S(M_F)]$$

7 Inhabitation

We provide a theoretical semi-decision procedure for solving the relativized inhabitation problems for $\mathsf{C1}$ and $\mathsf{C2}$, which is a decision procedure for bounded variants of the inhabitation problem. The procedure underlies the optimized implementation in the (CL)S system discussed in Sec. 8. To explain the procedure we need a few definitions.

Definition 5. *A* path π *is a type of the form* $\pi ::= A \mid \Box\pi \mid \vartheta \to \pi$. *A type* ϑ *is called* organized, *if it is an intersection of paths, i.e.,* $\vartheta \equiv \bigcap_{i\in I} \pi_i$. *The* length *of a path* $\vartheta_1 \to \cdots \to \vartheta_n \to \varrho$ *(where* ϱ *is not a function type) is defined to be* n. *We let* $\|\vartheta\|$ *denote the maximal length of a path in* ϑ *(assuming* ϑ *is organized). For a type* $\vartheta \equiv \vartheta_1 \to \cdots \to \vartheta_n \to \varrho$ *we let* $arg_i(\vartheta) \equiv \vartheta_i$ *for* $1 \leq i \leq n$, *and we let* $tgt_n(\vartheta) \equiv \varrho$. *Finally,* $\mathbb{P}_m(\vartheta)$ *denotes the set of paths of length at least* m *in* ϑ *(assuming* ϑ *is organized).*

It is easy to see that any type ϑ is equal to a polynomially sized organized type [2], and, whenever convenient, we shall tacitly assume that types are organized. Figure 4 is a semi-decision procedure for the inhabitation problem $\mathscr{C};\mathscr{D} \vdash_{\mathsf{C2}} ? : \psi$. It adapts the results of [2] to modal intersection types, and its correctness follows from the path lemmas presented in [13]. We use the notation of [2] where CHOOSE and OR denote nondeterministic choice and FORALL denotes universal branching of an alternating Turing-machine (ATM) [14]. By restriction to

```
   Input :  𝒞, 𝒟, ϑ
1  loop :
2  IF (ϑ ∈ 𝕊₁)THEN
3     CHOOSE (X : φ) ∈ 𝒞
4     CHOOSE 𝒮 ⊆fin 𝕍 → 𝕋₀ ∪ 𝔖₁ ∪ 𝔖₂
5     φ′ := ⋂{S(φ) | S ∈ 𝒮}
6     CHOOSE m ∈ {0, ..., ‖φ′‖};
7     CHOOSE P ⊆ ℙm(φ′);
8     IF (⋂π∈P tgtm(π) ≤ ϑ) THEN
9        IF (m = 0) THEN ACCEPT;
10       ELSE
11             FORALL(i = 1 ... m)
12                ϑ := ⋂π∈P argi(π);
13             GOTO loop;
14 ELSE
15    IF (ϑ = □φ) THEN
16       GOTO case1 OR GOTO case2
17    ELSE GOTO case2
18    case1 :
19       ϑ := φ; GOTO loop
20    case2 :
21       CHOOSE (F : ψ) ∈ 𝒟;
22       CHOOSE 𝒮 ⊆fin 𝕍 → 𝕋₀ ∪ 𝔖₁ ∪ 𝔖₂
23       ψ′ := ⋂{S(ψ) | S ∈ 𝒮}
24       CHOOSE m ∈ {0, ..., ‖ψ′‖};
25       CHOOSE P ⊆ ℙm(ψ′);
26       IF (⋂π∈P tgtm(π) ≤ ϑ) THEN
27          IF (m = 0) THEN ACCEPT;
28          ELSE
29                FORALL(i = 1 ... m)
30                   ϑ := ⋂π∈P argi(π);
31                GOTO loop;
```

Fig. 4. ATM semi-decision procedure for $\mathscr{C}; \mathscr{D} \vdash_{\mathsf{C2}} ? : \vartheta$

monomorphic derivations [1] or by bounding the size of substitutions to depth k in derivations [2] the semi-decision procedure shown in Fig. 4 becomes a decision procedure (cf. Thm. 2).

The restriction to grounded types in Thm. 1 does not limit the theoretical expressive power of the inhabitation relation:

Theorem 2 (Complexity). *Inhabitation in* C1 *and* C2 *is* $(k+2)$-EXPTIME*-complete with bound* k *(as in [2]) and* EXPTIME*-complete in the monomorphic case (as in [1]).*

The proof of the theorem can be found in [13] which also contains comments explaining why the restriction has no theoretical impact on expressiveness (which may seem surprising).

8 Experiments with (CL)S

We implemented the presented framework in the context of the (CL)S tool[2] [3, 4], using F# and C#. The inhabitation algorithm is configured for the bound $k = 0$ [2], limiting type instantiations to atomic types or intersections of such. We implemented an optimized version of the ATM in Fig. 4 and an L2-interpreter. We used them to solve suitable inhabitation problems and to generate L1-programs from resulting inhabitants. We provide a first experimental evaluation of our implementation by discussing three examples. The experiments were conducted on a computer with 8 GB main memory, Intel Core i5 (2.66 GHz), and Windows 8, using the .NET-Framework 4.0. For reasons of space we cannot provide all details, here.[3] In particular, it is not possible to present the L2-implementations of all $\mathscr{D}$-combinators or the generated L1-code, confining discussion to a few interesting combinators. Note that we extend the type language of L1 with type constructors of arbitrary arity that do not distribute over $\cap$.

We first extend the L1-repository $\mathscr{C}$ introduced in Sec. 5 to the tracking-scenario discussed in [4, Fig. 8], allowing to project a tracked object to its coordinates, for example. Furthermore, we add the following L2-combinators with associated implementations to $\mathscr{D}$:[4]

$$\texttt{avgFun} : \square(\texttt{TrObj} \to \texttt{R} \cap \mathfrak{a} \cap \mathit{ms}) \to \square[\texttt{TrObj}] \to \square(\texttt{R} \cap \mathfrak{a} \cap \mathit{Avg} \cap \mathit{ms})$$
$$\texttt{dist} : \square((\texttt{R},\texttt{R}) \cap \mathit{Cart}) \to \square((\texttt{R},\texttt{R}) \cap \mathit{Cart}) \to \square(\texttt{R} \cap \mathit{dist})$$

Here, `avgFun` uses code of a function that extracts a measured real value (with semantic property $\mathfrak{a}$) from a tracked object and code of an array of such objects to produce code of a real that is an average. Similarly, `dist` calculates distances between two coordinates. Using (CL)S, we solved various inhabitation questions for this scenario. For example, $\texttt{dist}(\texttt{box}\ \texttt{cdn}(\texttt{pos}(\texttt{TrV}(\texttt{0}))), \texttt{box}\ \texttt{cdn}(\texttt{pos}(\texttt{TrV}(\texttt{0}))))$ solves $\mathscr{C}; \mathscr{D} \vdash_{\texttt{C2}}? : \square(\texttt{R} \cap \mathit{dist})$ and has L1-code of type real describing a distance between objects. All synthesis-requests of this form were answered in $\leq$250ms.

Second, we consider $\mathscr{C}_2$ and $\mathscr{D}_2$ for synthesizing sorting routines for arrays of objects with a given order relation. Assume $\mathscr{C}_2$ only contains an L1-combinator $\texttt{lessThan} : ((\texttt{R},\texttt{R}) \to \texttt{bool}) \cap \mathit{incTO}$ deciding the standard total order $\leq_{\mathbb{R}}$ on $\mathbb{R}$, where incTO is a semantic type stating that $\leq_{\mathbb{R}}$ is an increasing total order. The repository $\mathscr{D}_2$ contains the combinators with associated implementations:

$$\begin{aligned}
\texttt{S} \quad &: \big(\square((\tilde{\alpha},\tilde{\alpha}) \to \texttt{bool}) \to \square([\tilde{\alpha}] \to [\tilde{\alpha}])\big) \ \cap\ \big(\square\mathfrak{a} \to \square(\top \to \mathfrak{a} \cap \mathit{Sorted})\big)\\
\texttt{swap} &: \big(\square((\tilde{\alpha},\tilde{\beta}) \to \gamma) \to \square((\tilde{\beta},\tilde{\alpha}) \to \gamma)\big) \ \cap\ \big(\square\mathfrak{a} \to \square\mathit{Rev}(\mathfrak{a})\big)\\
\Phi \quad &: \big(\square((\tilde{\alpha},\tilde{\alpha}) \to \texttt{bool}) \to \square((\tilde{\alpha},\tilde{\alpha}) \to \texttt{bool})\big) \cap\\
&\quad \big(\square\mathit{Rev}(\mathit{incTO}) \to \square\mathit{decTO}\big) \ \cap\ \big(\square\mathit{Rev}(\mathit{decTO}) \to \square\mathit{incTO}\big)
\end{aligned}$$

The combinator S contains an L1-template for bubble sort. Its first type component states that, given L1-code of a binary relation, S produces L1-code of

[2] http://www-seal.cs.tu-dortmund.de/seal/cls_en.shtml
[3] We refer to [13] for a comprehensive discussion of the examples and generated code.
[4] $[\vartheta]$ is a unary type constructor representing an array of objects of type ϑ.

a function mapping an array into an array. The semantic (second) component expresses that S returns code of a function that sorts an array without any distinguishing properties (we introduce $\top$ as a top element for semantic types for this purpose) according to the semantic property of the relation. Thus, if the relation happens to be a decreasing total order (i.e., $\mathfrak{a}$ gets instantiated with *decTO*), then S returns code of a function that sorts a typed array in decreasing order. The combinator `swap` is the modal version of the λ-calculus combinator that swaps the arguments of a function with a semantic property expressing that it reverses the order of arguments (*Rev*). The type of the combinator Φ (a purely logical combinator whose implementation is the identity function) expresses the idea that the reversal of an increasing total order is decreasing, and vice versa. The inhabitation question $\mathscr{C}_2; \mathscr{D}_2 \vdash_{\mathsf{C2}}? : \Box([\mathtt{R}] \to [\mathtt{R}] \cap \mathit{decTO} \cap \mathit{Sorted})$ produces the following inhabitant: $\mathtt{S}(\Phi(\mathtt{swap}(\mathsf{box}\ \mathtt{lessThan})))$. Its $\longmapsto^*$-reduction results in a corresponding L1-sorting routine. The subterm $\Phi(\mathtt{swap}(\mathsf{box}\ \mathtt{lessThan}))$ creates an L1-function of the form `fn (x,y) : (R,R) => lessThan(y,x)`. Semantically, this function reverses an increasing order into a decreasing order. It is passed to the implementation of S as an argument. Asking the inhabitation question above in (CL)S and carrying out the $\longmapsto^*$-reduction produced a bubble sort-based implementation of a corresponding sorting routine (24 lines of L1-code) within 150ms. Changing the inhabitation question by replacing *decTO* by *incTO*, the algorithm produces a sorting routine in increasing order. In [13] we extended the repositories in various ways, e.g., we synthesized a sorting routine for topologically sorting nodes of a directed acyclic graph.

Our last example combines the previous two scenarios and highlights the power of our framework for exploiting compositional design and higher order abstraction in synthesizing a non-trivial L1-program. Our goal is to synthesize code of a function which, when given an array of tracked objects, first calculates the average temperature of the tracked objects. This requires the synthesis of the function $(\mathsf{box}\ \mathtt{tmp}) \bullet (\mathsf{box}\ \mathtt{Tr})$ of Sec. 5. This function is then passed to a higher order L2-combinator that uses it to produce code which, when given an array of tracked objects, calculates their average temperature. The average temperature is then used to produce a function that filters out all objects from the array whose temperature is below average. The remaining array is sorted in decreasing order (with regard to temperature).[5] We assume that $\mathscr{D}_3$ contains the combinator `filterAndSort` which will be the top-level combinator for the inhabitant realizing the desired function.[6] Amongst others `filterAndSort` requires an argument whose type is given by the following combinator:

$$\mathtt{largerThanAvg} : \Box([\tilde{\alpha}] \to \tilde{\beta} \cap \mathit{Cel} \cap \mathit{Avg}) \to \Box(((\tilde{\beta}, \tilde{\beta}) \to \mathtt{bool}) \cap \mathit{decTO}) \to \\ \Box[\tilde{\alpha}] \to \Box(\tilde{\alpha} \to \tilde{\beta} \cap \mathit{Cel}) \to \Box(\tilde{\alpha} \to \mathtt{bool})$$

This combinator requires code which calculates an average temperature of an array of objects of type α, code of a decreasing total order, code of an array

[5] For example, if the tracked objects are reefer containers it is necessary to take action on those containers first that are the furthest above average.

[6] A complete discussion of this example can be found in [13].

of objects of type α, and code of a function that returns the temperature of an object of type α. It calculates the average temperature of the objects in the array by using the function provided as a first argument. Then it uses the total order and the temperature-function to compare the temperature of an object to the average temperature, returning true if it is larger than the average. The question $\mathscr{C}_3; \mathscr{D}_3 \vdash_{\mathsf{C2}}? : \Box([\mathtt{TrObj}] \to [\mathtt{TrObj}] \cap \mathit{decTO} \cap \mathit{Sorted})$ resulted in:

$$\begin{aligned}&\mathtt{filterAndSort}((\mathsf{box}\ \mathtt{tmp}) \bullet (\mathsf{box}\ \mathtt{TrV}),\\ &\quad \Phi(\mathtt{swap}(\mathsf{box}\ \mathtt{lessThan})),\mathtt{avgFun},\mathtt{largerThanAvg},\mathtt{F})\end{aligned}$$

As can be seen, `largerThanAvg` is an argument for `filterAndSort`. There is an interesting interaction between these two combinators. The L2-implementation of `filterAndSort` uses its *other* arguments to compute code with types of the arguments required by `largerThanAvg`. The L2-implementation of `filterAndSort` binds this code to names then passed to `largerThanAvg`. Thus, `largerThanAvg` indirectly uses functionality created by higher-order applications occurring in `filterAndSort` even though `largerThanAvg` exists *outside* `filterAndSort`. This is possible because `filterAndSort` takes `largerThanAvg` as argument and can thus provide it with bindings. The time required for synthesis and $\longmapsto^*$-reduction was approximately 9s and resulted in 61 lines of L1-code.

We conclude the discussion of (CL)S by mentioning an important principle for optimization of the inhabitation algorithm. Line 5 of the ATM (Fig. 4) indicates that the complexity arises from the construction of *all* possible type substitutions. Thus, one possible heuristic for optimization is to reduce the number of substitutions that actually have to be constructed. Using the fact that inhabitants must be well typed in L1 (cf. Thm. 1) the number of relevant type substitutions can be drastically decreased. This principle was used to optimize the inhabitation algorithm of (CL)S and showed major impact on runtime. The above initial experiments with SCS are encouraging, but further experiments, optimization heuristics, and engineering are needed.

9 Related Work

The idea of a staged approach to component-oriented synthesis does not appear to have been considered before. Our development of SCS would not, however, have been possible without the benefit of the modal analysis by Davies and Pfenning [10] of staged computation and their calculus $\lambda_e^{\Box\to}$. Not only can we transfer results from $\lambda_e^{\Box\to}$ to ensure semantic correctness (eliminability), but, interestingly, it turns out that modal types constitute a perfect instrument for exposing both the language- and phase distinction of staged computation to synthesis in combinatory logic.

Composition synthesis based on combinatory logic [6] with intersection types [7] was introduced and developed in [1–5]. The (CL)S-tool has been under development since 2011 and has been applied in several application scenarios, including generation of GUI and of control programs for LegoNXT robots [3].

Several optimizations have been implemented in the tool, including optimizations based on DFS-look-ahead strategies with subtype matching [5].

Composition synthesis is in deep accord with recent movements, in technically quite different branches of synthesis, towards component-orientation, where synthesis is considered relative to a given library of components (rather than construction from scratch) [15]. Our approach can be broadly compared in spirit (rather than in technology) to synthesis of loop free programs [16]. The combinatory approach is fundamentally different, at a technical level, from such approaches that are based on either temporal logic, automata theory, or traditional program logics.

Our approach is related to adaptation synthesis via proof counting [8, 9], where semantic types are combined with proof search in a specialized proof system. In particular, we follow [8, 9] in using semantic specifications at the interface level. The idea of adaptation synthesis [8] 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 (in [8] the specification language used is a typed predicate logic). Semantic intersection types can be compared to refinement types [17], but semantic types do not need to stand in a refinement relation to implementation types (as can be seen from our examples, this is important). Still, refinement types are a great source of inspiration for how semantic types can be used in specifications in many interesting situations.

10 Conclusion

We have introduced a framework for SCS based on modal intersection type systems and inhabitation in combinatory logic, and we have provided a theory of its correctness. The framework has been implemented in a prototype extension of the (CL)S system and has been used in experiments with SCS. Further work includes optimizations of the algorithm, in particular by exploiting the conservative extension property, more experimentation, and applications.

Acknowledgement. We thank our reviewers for very helpful reviews.

A Native Language L1

The native template language L1 is simultaneously defined and typed by the system shown in Fig. 5 below. It is a simply typed first order imperative core language with local references, extended with template variables u. The type structure consists of a set $\mathbb{T}_0$ of *value types* ranged over by t_0, reference types t_1 and the set of *native template types* $\mathbb{T}_1$, ranged over by τ. Value types are type variables ranged over by $\alpha, \beta, \gamma, \ldots$ drawn from the set $\mathbb{V}$, or type constants b including $*$ (unit type), `bool`, `int` and `real`.

$$\mathbb{T}_0 \ni t_0 ::= \alpha \mid b \qquad \mathbb{T}_0 \ni t_1 ::= \texttt{ref}\ t_0 \qquad \mathbb{T}_1 \ni \tau ::= t_0 \mid t_0 \to t_0$$

Native *program variables*, disjoint from template variables u, are ranged over by $\mathtt{x}$, and t ranges over all types (of the kind t_0, t_1, or τ). Judgements have the form $\Delta; \Sigma \vdash_{\text{L1}} T : t$, where the environment Δ contains bindings $(u : \tau)$ of template variables, and the environment Σ contains bindings $(\mathtt{x} : t)$ of program variables. *Native expressions* are template expressions T such that $\emptyset; \Sigma \vdash_{\text{L1}} T : t$ for some Σ and t. That is, native expressions do not contain any free template variables. *Native programs* are template expressions T such that $\emptyset; \emptyset \vdash_{\text{L1}} T : \tau$ for some τ. That is, native programs are closed expressions with no free variables and with types in $\mathbb{T}_1$. We assume in rule (cnst) further program constants c_t including the constant `ref` for creating references.[7]

$$\frac{}{\Delta; (\Sigma, \mathtt{x} : t) \vdash_{\text{L1}} \mathtt{x} : t}(\text{var}) \qquad \frac{}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{c}_t : t}(\text{cnst})$$

$$\frac{}{(\Delta, u : \tau); \Sigma \vdash_{\text{L1}} u : \tau}(\text{mvar}) \qquad \frac{}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{skip} : *}(\text{skip})$$

$$\frac{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{x} : \mathtt{ref}\ t_0}{\Delta; \Sigma \vdash_{\text{L1}} !\mathtt{x} : t_0}(\text{rd}) \qquad \frac{\Delta; \Sigma \vdash_{\text{L1}} T : t_0 \quad \Delta; \Sigma \vdash_{\text{L1}} \mathtt{x} : \mathtt{ref}\ t_0}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{x} := T : *}(\text{wr})$$

$$\frac{\Delta; \Sigma \vdash_{\text{L1}} T : \mathtt{bool} \quad \Delta; \Sigma \vdash_{\text{L1}} T_1 : t_0 \quad \Delta; \Sigma \vdash_{\text{L1}} T_2 : t_0}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{if}\ T\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 : t_0}(\text{if}) \qquad \frac{\Delta; \Sigma \vdash_{\text{L1}} T : \mathtt{bool} \quad \Delta; \Sigma \vdash_{\text{L1}} T_1 : *}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{while}\ T\ \mathtt{do}\ T_1 : *}(\text{wh})$$

$$\frac{\Delta; \Sigma \vdash_{\text{L1}} T_1 : * \quad \Delta; \Sigma \vdash_{\text{L1}} T_2 : t_0}{\Delta; \Sigma \vdash_{\text{L1}} T_1; T_2 : t_0}(\text{seq}) \qquad \frac{\Delta; \Sigma \vdash_{\text{L1}} T_1 : t \quad \Delta; (\Sigma, \mathtt{x} : t) \vdash_{\text{L1}} T_2 : t_0}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{let}\ \mathtt{x} : t = T_1\ \mathtt{in}\ T_2 : t_0}(\text{let})$$

$$\frac{\Delta; (\Sigma, \mathtt{x} : t_0) \vdash_{\text{L1}} T : t_0'}{\Delta; \Sigma \vdash_{\text{L1}} \mathtt{fn}\ \mathtt{x} : t_0 \Rightarrow T\ : t_0 \to t_0'}(\text{fn}) \qquad \frac{\Delta; \Sigma \vdash_{\text{L1}} T_1 : t_0 \to t_0' \quad \Delta; \Sigma \vdash_{\text{L1}} T_2 : t_0}{\Delta; \Sigma \vdash_{\text{L1}} (T_1\ T_2) : t_0'}(\to\text{E})$$

Fig. 5. Native template language `L1`

The following lemma can be proven as in [10] by induction on a derivation of the typing judgement. Notice the restriction to an empty environment Σ in the first assumption of the second property (see [10]).

Lemma 1 (Substitution).

1. *If* $\Delta; \Sigma \vdash_{\text{L1}} T : t$ *and* $\Delta; (\Sigma, \mathtt{x} : t) \vdash_{\text{L1}} T' : t'$ *then* $\Delta; \Sigma \vdash_{\text{L1}} T'[\mathtt{x} := T] : t'$.
2. *If* $\Delta; \emptyset \vdash_{\text{L1}} T : \tau$ *and* $(\Delta, u : \tau); \Sigma \vdash_{\text{L1}} T' : t$ *then* $\Delta; \Sigma \vdash_{\text{L1}} T'[u := T] : t$.
3. *Let* $S : \mathbb{V} \to \mathbb{T}_0$. *If* $\Delta; \Sigma \vdash_{\text{L1}} T : \tau$ *then* $S(\Delta); S(\Sigma) \vdash_{\text{L1}} S(T) : S(\tau)$.

[7] Reference types `ref` t_0 cannot escape local scopes by the type rules of `L1`, which simplifies our theory of intersection types (Sec. 3) which are unsound in the presence of unrestricted references [18]. The restriction can be lifted in several ways [18–20], but for brevity we shall not do so here.

B Operational Semantics of L2

The language and type system of L2 is identical to the calculus $\lambda_e^{\Box\to}$ introduced by Davies and Pfenning in [10], only our level L1 is decoupled from L2 in that it is distinguished as a different language with a type system and semantics of its own, and we use only a fully boxed fragment of the type language in which L2-types σ are generated from boxed types of L1 (of the form $\Box\tau$).[8]

The operational semantics of L2 is exactly the reduction relation $\longmapsto$ (and its reflexive transitive closure $\longmapsto^*$) defined for $\lambda_e^{\Box\to}$ in [10]. Computation is generated by β-reduction and the letbox-reduction rule (called $\Box\beta$ in [10]):

$$\mathsf{letbox}\ u = \mathsf{box}\ T\ \mathsf{in}\ M \longmapsto M[u := T]$$

together with congruences with respect to all contexts except for the context $\mathsf{box}\ T$. The reduction of letbox-expressions substitute template expressions $\mathsf{box}\ T$ into template variables u in L2-expressions M. This rule allows L2-programs to perform code substitution into L1-code. However, a boxed expression can itself be the result of L2-computations, as captured in the congruence rule

$$\frac{M_1 \longmapsto M_1'}{\mathsf{letbox}\ u = M_1\ \mathsf{in}\ M_2 \longmapsto \mathsf{letbox}\ u = M_1'\ \mathsf{in}\ M_2}$$

as can the L2-expression into which substitution is performed:

$$\frac{M_2 \longmapsto M_2'}{\mathsf{letbox}\ u = M_1\ \mathsf{in}\ M_2 \longmapsto \mathsf{letbox}\ u = M_1\ \mathsf{in}\ M_2'}$$

Because the relation $\longmapsto$ is *not* a congruence with respect to box-expressions (reduction does not "go under" box) it is possible to semantically decouple L1-expressions under the box-operator from the language level L2 (the contents of such boxed expressions are treated as black boxes). We refer the reader to [10] for full details of the semantics.

The type system imposes a strict phase distinction, in that metalanguage terms and only such can be reduced under $\longmapsto$ in L2, and, by subject reduction, expressions cannot "go wrong" under reduction (for example, by applying a boxed term, or by unboxing an unboxed term). Subterm occurrences in the scope of a box-constructor are, in the parlance of [10], *persistent*, in that they cannot be executed under metalanguage (L2) reduction. Term occurrences other than persistent term occurrences are called *eliminable* [10]. It is shown in [10] (subject reduction, Thm. 4, and eliminability, Thm. 5) that one has:

1. If $\Delta; \Gamma \vdash_{\text{L2}} M : \sigma$ and $M \longmapsto^* M'$ then $\Delta; \Gamma \vdash_{\text{L2}} M' : \sigma$
2. If $\emptyset; \emptyset \vdash_{\text{L2}} M : \Box\tau$ and $M \longmapsto^* M'$ and M' is irreducible, then M' contains no eliminable term occurrences.

Lemma 2 (Substitution [10]).

1. If $\Delta; \emptyset \vdash_{\text{L2}} M : \sigma$ and $\Delta; (\Gamma, x : \sigma) \vdash_{\text{L1}} M' : \sigma'$ then $\Delta; \Gamma \vdash_{\text{L2}} M'[x := M] : \sigma'$
2. Let $S : \mathbb{V} \to \mathbb{T}_0$. If $\Delta; \Gamma \vdash_{\text{L2}} M : \sigma$ then $S(\Delta); S(\Gamma) \vdash_{\text{L2}} S(M) : S(\sigma)$.

[8] This restriction is not essential, but it simplifies our presentation.

References

1. Rehof, J., Urzyczyn, P.: Finite Combinatory Logic with Intersection Types. In: Ong, L. (ed.) TLCA 2011. LNCS, vol. 6690, pp. 169–183. Springer, Heidelberg (2011)
2. Düdder, B., Martens, M., Rehof, J., Urzyczyn, P.: Bounded Combinatory Logic. In: Proceedings of CSL 2012, Schloss Dagstuhl. LIPIcs, vol. 16, pp. 243–258 (2012)
3. Düdder, B., Garbe, O., Martens, M., Rehof, J., Urzyczyn, P.: Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI Synthesis. In: Proceedings of ITRS 2012 (2012)
4. Rehof, J.: Towards Combinatory Logic Synthesis. In: 1st International Workshop on Behavioural Types, BEAT 2013. ACM (January 22, 2013)
5. Düdder, B., Martens, M., Rehof, J.: Intersection Type Matching with Subtyping. In: Hasegawa, M. (ed.) TLCA 2013. LNCS, vol. 7941, pp. 125–139. Springer, Heidelberg (2013)
6. Hindley, J.R., Seldin, J.P.: Lambda-calculus and Combinators, an Introduction. Cambridge University Press (2008)
7. Barendregt, H., Coppo, M., Dezani-Ciancaglini, M.: A Filter Lambda Model and the Completeness of Type Assignment. Journal of Symbolic Logic 48(4), 931–940 (1983)
8. Haack, C., Howard, B., Stoughton, A., Wells, J.B.: Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 83–98. Springer, Heidelberg (2002)
9. Wells, J.B., Yakobowski, B.: Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 262–277. Springer, Heidelberg (2005)
10. Davies, R., Pfenning, F.: A Modal Analysis of Staged Computation. Journal of the ACM 48(3), 555–604 (2001)
11. Dezani-Ciancaglini, M., Hindley, R.: Intersection Types for Combinatory Logic. Theoretical Computer Science 100(2), 303–324 (1992)
12. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform Proofs as a Foundation for Logic Programming. Ann. Pure Appl. Logic 51(1-2), 125–157 (1991)
13. Düdder, B., Martens, M., Rehof, J.: A Theory of Staged Composition Synthesis (Extended Version). Technical Report 843, Faculty of Computer Science, TU Dortmund (2013), `http://www-seal.cs.tu-dortmund.de/seal/downloads/research/cls/TR843-SCS.pdf`
14. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the ACM 28(1), 114–133 (1981)
15. Lustig, Y., Vardi, M.Y.: Synthesis from Component Libraries. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 395–409. Springer, Heidelberg (2009)
16. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of Loop-free Programs. In: Proceedings of PLDI 2011, pp. 62–73. ACM (2011)
17. Freeman, T., Pfenning, F.: Refinement Types for ML. In: Proceedings of PLDI 1991, pp. 268–277. ACM (1991)
18. Davies, R., Pfenning, F.: Intersection Types and Computational Effects. In: ICFP, pp. 198–208 (2000)
19. Dezani-Ciancaglini, M., Giannini, P., Della Rocca, S.R.: Intersection, Universally Quantified, and Reference Types. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 209–224. Springer, Heidelberg (2009)
20. Dezani-Ciangaglini, M., Ronchi Della Rocca, S.: Intersection and Reference Types. Essays dedicated to Henk Barendregt on the occasion of his 60'th birthday, pp. 77–86 (2007)