A Theory of Staged Composition Synthesis (Extended Version)

Boris Düdder
Technical University of Dortmund
Department of Computer Science
boris.duedder@cs.tu-dortmund.de

Moritz Martens
Technical University of Dortmund
Department of Computer Science
moritz.martens@cs.tu-dortmund.de

Jakob Rehof
Technical University of Dortmund
Department of Computer Science
jakob.rehof@cs.tu-dortmund.de

Number: 843
October 2013
Composition synthesis is based on the idea of using inhabitation in combinatory logic with intersection types as a foundation for computing compositions from component repositories. Components implemented in a language $L_1$ are exposed to composition synthesis in the form of typed combinator symbols. In practice, it is useful to augment the collection of components in the implementation language $L_1$ with a collection of components implemented in a metalanguage $L_2$ in order to define, e.g., higher-level (possibly application specific) composition operators and combinators that manipulate $L_1$-code templates. By extending composition synthesis to encompass both object language ($L_1$) and metalanguage ($L_2$) combinators, composition synthesis becomes a powerful and flexible framework for the generation of $L_1$-program compositions.

In this report a framework for composition synthesis is provided, in which the execution of synthesized programs can be staged into composition-time code generation ($L_2$) and run-time execution ($L_1$). A system of modal intersection types is introduced into a combinatory composition language to control the distinction between $L_1$- and $L_2$-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 composition time (stage 1), leaving only well typed $L_1$-code for later run-time execution (stage 2). Our framework has been implemented, and we report on experiments.

This is an extended version of the corresponding conference paper. In particular we present proofs for the main results. Furthermore, the appendix includes a detailed discussion of experiments.
Composition synthesis [19, 10, 8, 18, 9] is based on the idea of using inhabitation in combinatory logic [15] with intersection types [1] 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, $\Gamma$, 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 term $e$ exists such that $\Gamma \vdash e : \tau$. An algorithm (or semi-algorithm) for solving the inhabitation problem searches for inhabitants and can be used to synthesize inhabitants $e$. Under the propositions-as-types correspondence, inhabitation is the question of provability in a Hilbert-style presentation of a propositional logic, where $\Gamma$ represents a propositional theory, $\tau$ represents a proposition to be proved, and $e$ is a proof.

Following [13, 20], 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 of [19, 10, 8, 18, 9] semantic types are represented by intersection types [1]. 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

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

native types (real, real $\times$ real, . . .) 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 report 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. In particular we present all details of the proofs underlying the theoretical results. 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 $L_1$ in which we can execute programs at runtime, referred to as the native language. Following the ideas summarized in [18], components written in $L_1$ can be exposed for composition synthesis through a combinatory environment $\mathcal{E}$ in which components are exposed as semantically typed combinator symbols $(X : \phi)$, where $X$ is the name of the component and $\phi^\circ$ is the native type of the component in the language $L_1$ (the map $(\cdot)^\circ$ erases semantic type information). So we assume for each $(X : \phi) \in \mathcal{E}$ that we have a native implementation program $T_X$ with $\vdash_{L_1} T_X : \phi^\circ$, where $\vdash_{L_1}$ formalizes the type system of $L_1$, and where the implementation $T_X$ is associated with the typed combinator symbol $(X : \phi)$. Composition of $L_1$-components from the environment $\mathcal{E}$ can be formalized in a corresponding combinatory logic, $C_1$. We take a simple monomorphic imperative first-order language as our exemplary native language $L_1$.

In our SCS framework we want to enhance our ability to compute compositions of $L_1$-programs by introducing templates of $L_1$ program fragments and expressions into which we can substitute other $L_1$-program expressions to build complex $L_1$-programs from simpler ones. To realize this idea in full, we need a possibly different language, $L_2$, referred to as the compositional metalanguage, which is suited for the metaprogramming tasks involved in computing over $L_1$-templates. Since a central task here is to perform substitutions into $L_1$-templates we take the $\lambda$-calculus as our exemplary $L_2$-language. In order to formalize this situation, we introduce type variables (type templates) and special program expression template variables $u$ into $L_1$ to serve as substitutable placeholders for $L_1$-expressions inside other $L_1$-expressions. Moreover, we introduce a type system $\vdash_{L_2}$ for the metalanguage $L_2$. Now, if we could compose both $L_1$-programs and $L_2$-programs that compose $L_1$-programs, we could achieve more flexible and powerful forms of composition, since we can implement special code-generating $L_1$-composition operators in $L_2$, depending on situation and purpose. This situation, in turn, can be formalized by introducing a combinatory logic, $C_2$, in which compositions of $L_2$-programs can be computed. In this combinatory logic $C_2$, implemented $L_2$-components would be exposed in combinatory environments $\mathcal{D}$ analogously to the way $L_1$-components are exposed in environments $\mathcal{E}$ in $C_1$. We now have two implementation languages, $L_1$ and $L_2$, exposed for combinatory composition through associated combinatory logics, $C_1$ and $C_2$. In this system, we need a phase distinction between composition time computations in $L_2$ and runtime computations in $L_1$: we first (stage 1) perform composition time computations in the metalanguage $L_2$ which produce $L_1$-programs to be executed at runtime (stage 2). Since our focus is entirely on the generation of $L_1$-compositions, we shall focus on $L_2$-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 [4], using modal types of the form □τ 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 □τ 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. Theorem 23), 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 Synthesis) tool, and we report on the results of experiments using the tool for SCS.

ORGANIZATION OF THE REPORT

The remainder of this report is organized as follows. In Sec. 2 we introduce the native language L1 and the metalanguage L2. 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 report. The appendix consists of four sections, giving a short introduction to our implemented tool (CL)S, a main presentation of (CL)S, the discussion of examples, respectively a short specification of the test system.
We introduce an exemplary native language $L_1$ and a compositional metalanguage $L_2$, referred to collectively as implementation languages. In distinction to the framework of Davies and Pfenning [4] 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 contrast, in the framework of [4] arbitrary levels of stages can be distinguished within a single language). 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 $\text{box } T$ to the language $L_2$ with $L_2$-types of the form $\Box \tau$ (where $\tau$ is an $L_1$-type), but other than that the theory of $L_2$ is agnostic of the nature of programs $T$ and types $\tau$ of $L_1$.

For concreteness, we fix a simply typed first-order core language as an exemplary native implementation core language, but (as mentioned) $L_1$ can be exchanged easily. The only requirements on the design of $L_1$ are that $L_1$ should be typed, it should contain functions and function application, the language should satisfy preservation of types under appropriate term substitution (see substitution Lemma 1 below for $L_1$), and that well typed $L_1$-programs can be executed at a later runtime stage (with which we are not further concerned in this report).

### 2.1 Native Language $L_1$

The native language consists of template expressions $T$ in which code templates using special template variables ranged over by $u$ can be formed into which other native expressions or templates can be substituted. The native template language $L_1$ is simultaneously defined and typed by the system shown in Figure 2.1. 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 $T_0$ of value types ranged over by $t_0$, reference types $t_1$ and the set of native template types $T_1$, ranged over by $\tau$. Value types are type constants $b$ including the unit type $\ast$, $\text{bool}$, $\text{int}$ and $\text{real}$. Template types may in addition contain type variables ranged over by $\alpha, \beta, \gamma, \ldots$ drawn from the set $\mathbb{V}$.

$$
T_0 \ni t_0 ::= \alpha \mid b \\
T_1 \ni t_1 ::= \text{ref } t_0 \\
T_1 \ni \tau ::= t_0 \mid t_0 \rightarrow t_0
$$

Native program variables, disjoint from template variables $u$, are ranged over by $x$, and $t$ ranges over all types (of the kind $t_0$, $t_1$, or $\tau$). Judgements have the form $\Delta, \Sigma \vdash_{L_1} T : t$, where
the environment $\Delta$ contains bindings $(u : \tau)$ of template variables, and the environment $\Sigma$ contains bindings $(x : t)$ of program variables. Native expressions are template expressions $T$ such that $\emptyset; \Sigma \vdash_{T_1} T : t$ for some $\Sigma$ and $t$. That is, native expressions do not contain any free template variables. Native programs are template expressions $T$ such that $\emptyset; \emptyset \vdash_{T_1} L_1 T : \tau$ for some $\tau$. That is, native programs are closed expressions with no free variables and with types in $T_1$. We assume in rule (cnst) further program constants $c_t$ including the constant $\text{ref} : t_0 \rightarrow \text{ref} t_0$ for creating references.

We do not specify an operational semantics for $L_1$, since it is altogether standard, and we are mainly concerned with computations in the metalanguage which we will consider next.

The following substitution lemma can be proven as in [4]. Notice the restriction to an empty environment $\Sigma$ in the first assumption of the second property (see [4]).

**Lemma 1 (Substitution)**

1. If $\Delta; \Sigma \vdash_{T_1} T : t$ and $\Delta; (\Sigma, x : t) \vdash_{T_1} T'[x := T] : t'$. 

2. If $\Delta; \emptyset \vdash_{T_1} T : \tau$ and $(\Delta, u : \tau); \Sigma \vdash_{T_1} T'[u := T] : t$.

**Proof:** Induction on the derivation of typings for $T'$. $\square$

We can substitute $T_0$-types for type variables.

**Lemma 2 (Type substitution)** Let $S$ be a type substitution $\forall \rightarrow T_0$. If $\Delta; \Sigma \vdash_{T_1} T : \tau$ then $S(\Delta); S(\Sigma) \vdash_{T_1} S(T) : S(\tau)$.

**Proof:** By induction on the derivation of the typing judgement. $\square$

### 2.2 Compositional Metalanguage $L_2$

The compositional metalanguage $L_2$ is a standard $\lambda$-calculus with simple types extended with modal types as introduced by Davies and Pfenning [4] to distinguish computational stages at the type level. In our setting, we can intuitively understand an $L_2$-type $\boxdot \tau$ ($\tau \in T_1$) as meaning "$L_1$-code with $L_1$-type $\tau$".

The set $T_2$ denotes metalanguage types, ranged over by $\sigma$. The modal type constructor $\boxdot$ is a special covariant constructor.

$$T_2 \ni \sigma ::= \boxdot \tau \mid \sigma \rightarrow \sigma'$$

1. Notice that reference types $\text{ref} t_0$ are restricted by the type rules of $L_1$ to be non-escaping: by the type rules, they cannot escape local scopes (defined by let-expressions, rule (let)), they cannot be passed as arguments, and they cannot escape functions (function types $\tau$ are in $T_1$ which contain no reference types, rule (fn)). Hence, expressions of type $\tau$ can be treated as functional. This restriction simplifies the development of our intersection type theory of semantic types (Chapter 3), since standard intersection types are unsound in the presence of unrestricted references [3], but the restriction can be lifted at the price of an adapted theory in several ways [3, 5, 7]. We shall not adopt any such here but rather restrict our type interfaces to be functional, for brevity.
\[
\begin{align*}
\text{\(\Delta; (\Sigma, x : t) \vdash_{L_1} x : t\)} & \quad (\text{var}) \\
\text{\(\Delta; \Sigma \vdash_{L_1} \epsilon_i : t\)} & \quad (\text{cnst}) \\
(\Delta, u : \tau); \Sigma \vdash_{L_1} u : \tau & \quad (\text{mvar}) \\
\Delta; \Sigma \vdash_{L_1} \text{skip} : * & \quad (\text{skip}) \\
\Delta; \Sigma \vdash_{L_1} x : \text{ref} t_0 & \quad (\text{rd}) \\
\Delta; \Sigma \vdash_{L_1} ! x : t_0 & \quad (\text{wr}) \\
\Delta; \Sigma \vdash_{L_1} T : \text{bool} & \quad (\text{if}) \\
\Delta; \Sigma \vdash_{L_1} \text{while} T \text{ do } T_1 : * & \quad (\text{wh}) \\
\Delta; \Sigma \vdash_{L_1} \text{let } x : t \text{ in } T_2 : t_0 & \quad (\text{let}) \\
\Delta; \Sigma \vdash_{L_1} \text{fn } x : t_0 = > T : t_0 \rightarrow t_0' & \quad (\text{fn}) \\
\Delta; \Sigma \vdash_{L_1} T_1 : t_0 \rightarrow t_0' & \quad (\rightarrow E)
\end{align*}
\]

Figure 2.1: Native template language \(L_1\)

Compositional metalanguage terms are terms of the \(\lambda^{\boxleftarrow}e\)-calculus \([4]\), ranged over by \(M\):

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

Compositional metalanguage expressions are typed by the system \(L_2\) shown in Figure 2.2. Judgements are of the form \(\Delta; \Gamma \vdash_{L_2} M : \sigma\), where \(\Delta\) contains \(L_1\)-bindings \((u : \tau)\) of native template variables to \(L_1\)-types, and \(\Gamma\) is the standard \(\lambda\)-calculus type environment of bindings \((x : \sigma)\) of \(\lambda\)-variables to \(L_2\)-types.

The rule \((\boxneg I)\) together with the environment \(\Delta\) provide the interface between \(L_1\) and \(L_2\). According to this rule, native templates \(T\) that are well typed with native template types in \(L_1\) can be injected into \(L_2\) by being placed in the scope of the \(\text{box}\)-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 \( L_2 \). As shown in [4], this discipline ensures that we can soundly substitute native expressions into native templates in \( L_2 \)-computations. The dual rule, \((\text{CE})\) discharges assumptions in the template variable \( \Delta \) using the letbox construct. As detailed in Section 2.3, this construct performs substitution of native templates into native template variables under \( L_2 \)-computation.

\[
\begin{align*}
\Delta; (\Gamma, x : \sigma) \vdash_{L_2} x : \sigma && \text{(var)} \\
\Delta; (\Gamma, x : \sigma) \vdash_{L_2} M : \sigma' && \text{(→I)} \\
\Delta; \Gamma \vdash_{L_2} \lambda x : \sigma. M : \sigma \rightarrow \sigma' && \text{(→E)} \\
\Delta; \Gamma \vdash_{L_2} (M_1 M_2) : \sigma' && \text{(-E)} \\
\Delta; \emptyset \vdash_{L_2} T : \tau && \text{(-I)} \\
\Delta; \Gamma \vdash_{L_2} \text{letbox } u : \tau = M_1 \text{ in } M_2 : \sigma && \text{(-E)} 
\end{align*}
\]

Figure 2.2: Metalanguage \( L_2 \)

The following substitution lemmas can be transferred directly from [4].

**Lemma 3 (Substitution)** If \( \Delta; \emptyset \vdash_{L_2} M : \sigma \) and \( \Delta; (\Gamma, x : \sigma) \vdash_{L_1} M' : \sigma' \) then \( \Delta; \Gamma \vdash_{L_2} M'[x := M] : \sigma' \)

**Lemma 4 (Type substitution)** Let \( S \) be a type substitution \( \forall \rightarrow \bigwedge \). If \( \Delta; \Gamma \vdash_{L_2} M : \sigma \) then \( S(\Delta); S(\Gamma) \vdash_{L_2} S(M) : S(\sigma) \).

### 2.3 Operational Semantics

As mentioned above, we consider \( L_2 \)-reduction as composition-time computation (stage 1), and we consider computation in \( L_1 \) (not detailed here) as run-time execution which is deferred to stage 2 after composition-time computation is complete. Thus, the goal of \( L_2 \)-computation is to compute well typed expressions of \( L_1 \) that can later be executed according to \( L_1 \)-semantics. For this to work out, it must be ensured that only \( L_2 \)-expressions can be executed at composition-time, that all such expressions can be computed away completely in any term whose type (of form \( \square \tau \)) indicates that the result is a boxed \( L_1 \)-program, and that only well-typed \( L_1 \)-programs can result. The type system \( L_2 \) accomplishes these goals.
The language and type system of $L_2$ is identical to the calculus $\lambda \rightarrow \Box$ introduced by Davies and Pfenning in [4], only our level $L_1$ is decoupled from $L_2$ 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 $L_2$-types $\sigma$ are generated from boxed types of $L_1$ (of the form $\Box \tau$).\(^2\)

The operational semantics of the language $L_2$ is exactly the reduction relation $\rightarrow$ (and its reflexive transitive closure $\rightarrow^*$) defined for $\lambda \rightarrow \Box e$ in [4]. Computation is generated by $\beta$-reduction and the letbox-reduction rule (called $\Box \beta$ in [4]):

$$\text{letbox } u = \text{box } T \text{ in } M \rightarrow M[u := T]$$

together with congruences with respect to all contexts except for the context $\text{box } T$. The reduction of letbox-expressions substitute template expressions $\text{box } T$ into template variables $u$ in $L_2$-expressions $M$. This rule allows $L_2$-programs to perform code substitution into $L_1$-code. However, a boxed expression can itself be the result of $L_2$-computations, as captured in the congruence rule

$$\frac{M_1 \rightarrow M_1'}{\text{letbox } u = M_1 \text{ in } M_2 \rightarrow \text{letbox } u = M_1' \text{ in } M_2}$$

as can the $L_2$-expression $M$ into which substitution is performed:

$$\frac{M_2 \rightarrow M_1'}{\text{letbox } u = M_1 \text{ in } M_2 \rightarrow \text{letbox } u = M_1 \text{ in } M_2'}$$

Because the relation $\rightarrow$ is not a congruence with respect to box $\Box$-expressions (reduction does not “go under” box $\Box$) it is possible to semantically decouple $L_1$-expressions under the box-operator from the language level $L_2$ (the contents of such boxed expressions are treated as black boxes). We refer the reader to [4] 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 $\rightarrow$ in $L_2$, 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 $\Box$-constructor are, in the parlance of [4], persistent, in that they can not be executed under metalanguage ($L_2$) reduction.

**Theorem 5** (Subject reduction [4], Theorem 4) If $\Delta; \Gamma \vdash_{L_2} M : \sigma$ and $M \rightarrow^* M'$ then $\Delta; \Gamma \vdash_{L_2} M' : \sigma$

Term occurrences other than persistent term occurrences are called eliminable [4], and we have:

**Theorem 6** (Eliminability [4], Theorem 5) If $\emptyset; \emptyset \vdash_{L_2} M : \Box \tau$ and $M \rightarrow^* M'$ and $M'$ is irreducible, then $M'$ contains no eliminable term occurrences.

\(^2\) This restriction is not essential, but it simplifies our presentation.
As a consequence of the theorems above, typability in system L2 implies that reducing, in L2, an expression of type □τ to normal (irreducible) 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 □τ 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).
The sets $\mathcal{E}_i$ of semantic types of level $i$ ($i = 1, 2$) are ranged over by $t$ and $s$, respectively. Type variables of level 1 are ranged over by $a \in V_{\mathcal{E}_1}$, and type variables of level 2 are ranged over by $b \in V_{\mathcal{E}_2}$. We assume that $V$, $V_{\mathcal{E}_1}$, and $V_{\mathcal{E}_2}$ are disjoint sets. We assume sets of semantic type constants $D_1$ and $D_2$, with $D_1$ and $D_2$ disjoint from each other and from the constants of $L_1$.

The set $\mathcal{E}_1$ contains a copy $^1$ of $T_1$ built from distinct sets of variables $V_{\mathcal{E}_1}$ and constants $D_1$ and closed under intersection. The set $\mathcal{E}_2$ is built analogously, as a copy of $T_2$ over distinct variables and constants $V_{\mathcal{E}_2}$ and constants $D_2$:

$\mathcal{E}_1 \ni t ::= a | d_1 \mid t' \mid t \cap t'$

$\mathcal{E}_2 \ni s ::= b | d_2 \mid s' \mid s \cap s' \mid \Box t$

The set of semantic $L_1$-types $S_1$ is ranged over by $\phi$, and the set of semantic $L_2$-types $S_2$ is ranged over by $\psi$:

$S_1 \ni \phi ::= \tau | \phi \cap t | t \cap \phi | \phi \rightarrow \phi' | \phi \cap \phi'$

$S_2 \ni \psi ::= \sigma | \psi \cap s | s \cap \psi | \Box \phi | \phi \rightarrow \psi' | \psi \cap \psi'$

Type expressions are implicitly considered as equivalence classes modulo commutativity, associativity and idempotency of $\cap$.

We let $\vartheta, \varrho, \upsilon$ range over $S_1 \cup S_2 \cup \mathcal{E}_1 \cup \mathcal{E}_2$. Notice that $T_1$ and $T_2$ are disjoint, $S_1$ and $S_2$ are disjoint, and $T_i \subseteq S_i$. An atom is a type variable or a type constant, and we let $A$ range over atoms.

The semantic type structures $S_i$ are constructed according to a particular method. Semantic types $\mathcal{E}_i$ are defined as a separate kind of types containing “copies” of the implementation type languages $T_i$, and the former are superimposed onto the latter in $S_i$, respectively, by “attaching” types in $\mathcal{E}_i$ to types in $T_i$ with intersections. This construction allows for maximal freedom in combining semantic types with “underlying” (see definition 10) implementation types, thereby achieving more generality than, for example, refinement types [11], while at the same time allowing us to treat the semantic types as a distinct kind from implementation types. Thus, the type structures $T_0$ and $\mathcal{E}_i$ are treated as different kinds in the following definition of substitution, which turns out to be technically expedient in theory as well as important for efficiency later, when we consider inhabitation.

1 Function types of $\mathcal{E}_1$ and $S_1$ below are not restricted to be first order, since it is developed for the general case. If $L_1$ happens to be restricted to a first-order system, as in our example case, our semantic type framework is more general.
Definition 7 A type substitution is a map
\[ S : \mathcal{V} \cup \mathcal{V}_1 \cup \mathcal{V}_2 \rightarrow \mathcal{T}_0 \cup \mathcal{S}_1 \cup \mathcal{S}_2 \]
satisfying the following conditions: \( \forall a \in \mathcal{V}, S(a) \in \mathcal{T}_0, \forall a \in \mathcal{V}_1, S(a) \in \mathcal{S}_1, \) and \( \forall b \in \mathcal{V}_2, S(b) \in \mathcal{S}_2. \)

Lemma 8 Let \( S \) be a type substitution and let \( Q \) be any one of the sets \( \mathcal{T}_i, \mathcal{S}_i, \) or \( \mathcal{S}_i \) \((i = 1, 2)\).
Whenever \( \vartheta \in Q \), we have \( S(\vartheta) \in Q. \)

The following definition is standard for intersection types [1].

Definition 9 Subtyping \( \leq \) is the least preorder (reflexive and transitive relation) on \( \mathcal{S}_1 \cup \mathcal{S}_2 \cup \mathcal{S}_1 \cup \mathcal{S}_2 \), satisfying the following conditions:

- \( \vartheta \cap \varphi \leq \vartheta, \vartheta \cap \nu \leq \nu \)
- \( (\vartheta \rightarrow \varphi) \cap (\vartheta \rightarrow \nu) \leq \vartheta \rightarrow \varphi \cap \nu \)
- \( \vartheta \leq \vartheta' \wedge \varphi \leq \varphi' \Rightarrow \vartheta \rightarrow \varphi \leq \vartheta' \rightarrow \varphi' \)
- \( \vartheta \leq \vartheta' \wedge \varphi \leq \varphi' \Rightarrow \vartheta \cap \varphi \leq \vartheta' \cap \varphi' \)
- \( (\Box \vartheta) \cap (\Box \varphi) \leq (\Box (\vartheta \cap \varphi)) \)
- \( \vartheta \leq \varphi \Rightarrow \Box \vartheta \leq \Box \varphi \)

We identify \( \vartheta \) and \( \varphi \), written \( \vartheta = \varphi \), when \( \vartheta \leq \varphi \) and \( \varphi \leq \vartheta \). The relation \( = \) is referred to as type equality. We write \( \vartheta \equiv \varphi \), if \( \vartheta \) and \( \varphi \) are identical as syntax trees.

Notice that the following distributivity properties follow from the axioms of subtyping:

- \( (\vartheta \rightarrow \varphi) \cap (\vartheta \rightarrow \nu) = \vartheta \rightarrow \varphi \cap \nu \)
- \( (\vartheta \rightarrow \varphi) \cap (\vartheta' \rightarrow \varphi') \leq (\vartheta \cap \vartheta') \rightarrow (\varphi \cap \varphi') \)
- \( (\Box \vartheta) \cap (\Box \varphi) = \Box (\vartheta \cap \varphi) \)

We tacitly assume that the semantic type structures \( \mathcal{D}_1 \) and \( \mathcal{D}_2 \) can be equipped with partial orders \( \leq_{\mathcal{D}_i} \) in which case axioms \( d_i \leq_{\mathcal{D}_i} d'_i \Rightarrow d_i \leq d'_i \) are added to the axiomatization of subtyping.
COMBINATORY LOGIC

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 [6], and the rules of C2 are extended (in rule (□I)) according to the modal extension of L2.

Combinatory C1-terms are defined by

\[ e ::= X \mid (e_1 e_2) \]

Environments \( \mathcal{C} \) are finite sets of bindings of the form \((X : \phi)\) with \(\phi \in S_1\). The combinatory logic C1 is defined by the rules of Figure 4.1.

\[
\begin{align*}
\mathcal{C}, X : \phi & \vdash_{c_1} X : S(\phi) & (\text{var}) \\
\mathcal{C} & \vdash_{c_1} e_1 : \phi \rightarrow \phi' & \mathcal{C} & \vdash_{c_1} e_2 : \phi (\rightarrow E) \\
\mathcal{C} & \vdash_{c_1} e : \phi & \mathcal{C} & \vdash_{c_1} e : \phi' (\cap I) \\
\mathcal{C} & \vdash_{c_1} e : \phi \cap \phi' & (\cap) & \mathcal{C} \vdash_{c_1} e : \phi' (\leq) & \mathcal{C} \vdash_{c_1} e : \phi \leq \phi' & (\leq)
\end{align*}
\]

Figure 4.1: Combinatory logic C1

Combinatory C2-terms are defined by

\[ E ::= F \mid (E_1 E_2) \mid \box e \]

Environments \( \mathcal{D} \) are finite sets of bindings of the form \((F : \psi)\) with \(\psi \in S_2\). The combinatory logic C2 is defined by the rules of Figure 4.2.

4.1 INHABITATION AND COMPOSITION SYNTHESIS

In composition synthesis (see [18] for a general introduction), we are concerned with the relativized inhabitation problem: Given \( \mathcal{C}, \mathcal{D} \) and \(\psi\), does there exist a combinatory term \(E\) such that \(\mathcal{C}; \mathcal{D} \vdash_{c_2} E : \psi\)? We use the notation \(\mathcal{C}; \mathcal{D} \vdash_{c_2}? : \psi\) 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 \(\mathcal{C}\) and \(\mathcal{D}\) as component repositories and \(\psi\) as a synthesis goal specification.
It should be noted that the expressive power of the unrestricted inhabitation relation is enormous: it is undecidable even in simple types.\textsuperscript{1} We can indeed consider the inhabitation relation as an operational semantics for a Turing-complete abstract logic programming language\textsuperscript{[17]} at the level of interface types, as suggested in\textsuperscript{[18]}. Or, we can restrict the relation to ensure that search for inhabitants always terminates. We consider algorithmic aspects of the relation in Chapter 7.

\textsuperscript{1} Note that the combinatory inhabitation relation considered here is not to be confused with inhabitation in $\lambda$-calculus, where inhabitation is, e.g., \textsc{pspace}-complete for simple types by Statman’s theorem. Our relation is much more expressive, because it is not confined to a fixed base but is relativized to arbitrary environments $\mathcal{C}, \mathcal{D}$ (\textsuperscript{18} contains a survey of the relevant results).
EXAMPLE

We introduce a simple example adapted from [18] and extended with our modal types to illustrate a few basic features of the formal system. For ease of reading, we write $\mathcal{S}_1$-types in blue font (as in $\text{Cel}$) and $\mathcal{S}_2$-types in red font (as in $\text{Conv}$). Programs and combinators in $L_1$ are written in green typewriter font. Whenever convenient we use the shorthand notation $\tilde{\tau}$ to denote an $S_1$-type consisting of the $L_1$-type $\tau$ intersected with an associated semantic type variable $a_{\tau}$ from $\mathcal{S}_1$, so, for example, $\tilde{\alpha}$ denotes the type $\alpha \cap a_{\alpha}$. $L_1$-types are written in black typewriter font, for example, $R$ which denotes the type of reals.

Let $\mathcal{C}$ contain the combinators (we freely extend the $L_1$-type language by type constructors)

\[
\begin{align*}
0 & : \text{TrObj} \\
\text{Tr} & : \text{TrObj} \to D((R, R) \cap \text{Cart}, R \cap \text{Gpst}, R \cap \text{Cel}) \\
\text{tmp} & : D((R, R), R, R \cap a) \to R \cap a \cap ms
\end{align*}
\]

The environment $\mathcal{C}$ could be part of a semantic repository of components to track (Tr) an object (0) 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 $\text{tmp}$ 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 $\mathcal{D}$ contain the combinators

\[
\begin{align*}
\bullet & : \Box(\tilde{\beta} \to \tilde{\gamma}) \to \Box(\tilde{\kappa} \to \tilde{\beta}) \to \Box(\tilde{\kappa} \to \tilde{\gamma}) \\
c12fh & : (\Box(R \cap \text{Cel}) \to \Box(R \cap \text{Fh})) \cap \text{Conv} \\
\Diamond & : \Box(\tilde{\alpha} \cap ms) \to (\Box\tilde{\alpha} \to \Box\tilde{\beta}) \cap \text{Conv} \to \Box(\tilde{\beta} \cap ms)
\end{align*}
\]
with the following bindings to implementations in L2:

- $\triangleq \lambda G : \square(\beta \rightarrow \gamma). \lambda F : \square(\alpha \rightarrow \beta).

  \begin{align*}
  & \text{letbox } f : \alpha \rightarrow \beta = F \text{ in} \\
  & \text{letbox } g : \beta \rightarrow \gamma = G \text{ in} \\
  & \text{box} \\
  & \quad (\text{fn } y : \alpha = => (g (f y)))
  \end{align*}

- $\triangleq \lambda z : \square R.

  \begin{align*}
  & \text{letbox } u : R = z \text{ in} \\
  & \text{box} \\
  & \quad \text{let } x : R = u \text{ in} \\
  & \quad \quad (x \ast (9 \div 5) + 32)
  \end{align*}

- $\triangleq \lambda z : \square \alpha. \lambda F : \square \alpha \rightarrow \square \beta.

  (F z)$

The semantic $\Sigma_2$-type $\text{Conv}$ of the combinator $\text{cl2fh}$ expresses the idea that the corresponding function acts purely as a unit conversion. The type of the combinator $\diamond$ uses this type and the $\Sigma_1$-type $\text{ms}$ to express the idea that a conversion can be applied to a measurement to produce something which is still a measurement.

Suppose we wish to know whether it is possible to compose a function from the component repositories $\mathcal{C}$ and $\mathcal{D}$ which measures the temperature in Celsius of an object. We can formalize this query as the inhabitation problem

$$\mathcal{C} ; \mathcal{D} \vdash \mathcal{C}_2 : \square (\text{TrObj} \rightarrow (R \cap Cel \cap ms))$$

which has the solution (writing $\bullet$ in infix notation):

$$(\text{box } \text{tmp}) \bullet (\text{box } \text{Tr}) : \square (\text{TrObj} \rightarrow (R \cap Cel \cap ms))$$

Performing the L2-reduction

$$(\text{box } \text{tmp}) \bullet (\text{box } \text{Tr}) \mapsto^* \text{box} (\text{fn } y : \text{TrObj} = => (\text{tmp } (\text{Tr } y)))$$

we see that L1-code implementing such a function is produced.

If we ask for $\mathcal{C} ; \mathcal{D} \vdash \mathcal{C}_2 : \square (R \cap Fh \cap ms)$, the solution is (writing $\diamond$ in infix notation):

$$(\text{box } (\text{tmp } (\text{Tr } 0))) \diamond \text{cl2fh}$$

with the L2-reduction

$$(\text{box } (\text{tmp } (\text{Tr } 0))) \diamond \text{cl2fh} \mapsto^* \text{box}$$

$$\quad \text{let } x : R = \text{tmp } (\text{Tr } 0) \text{ in}$$

$$\quad \quad (x \ast (9 \div 5) + 32)$$
If we add the combinator \( c2f : (R \cap Cel \cap ms) \rightarrow (R \cap Fh \cap ms) \) to \( C \), we would get the additional solution for the same inhabitation goal:

\[
\text{box} (c2f (\text{tmp} \ (\text{Tr} \ 0)))
\]

If, in addition, we add to \( \mathcal{D} \) the modal apply combinator

\[
mapply : \Box(\alpha \rightarrow \beta) \rightarrow \Box\alpha \rightarrow \Box\beta
\]

with definition

\[
mapply \triangleq \lambda F : \Box(\alpha \rightarrow \beta), \lambda z : \Box\alpha. \\
\text{letbox} \ f : \alpha \rightarrow \beta = F \text{ in} \\
\text{letbox} \ u : \alpha = z \text{ in box} (f \ z)
\]

we get also the solution

\[
mapply (\text{box} \ c2f) (\text{mapply} ((\text{box} \ \text{tmp}) \bullet (\text{box} \ \text{Tr}))(\text{box} \ 0))
\]

reducing in \( L2 \) to the \( L1 \)-program

\[
\text{box} (c2f ((\text{fn} \ y : \text{TrObj} \Rightarrow (\text{tmp} \ (\text{Tr} \ y))) \ 0))
\]

As can be seen that from these examples, the inhabitation relation determines the possible placements of the box-constructor, in each case determining a specific “division of labour” between \( L1 \) and \( L2 \). It should also be evident that 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 (CLS) as discussed in Chapter 7 and Chapter 8, and further examples are given in Chapters 8 and C.
Our main result is a conservative extension theorem (Theorem 23) showing that combinatory compositions performed over restricted (supported or grounded, Definition 18) environments can be transformed into well typed implementation language expressions. In this chapter we present the technical proof of this theorem. A major step towards the result, in turn, is an embedding property (Proposition 19). The remainder of this section is organized as follows. Section 6.1 contains some definitions and lemmas pertaining to types and subtyping needed for the embedding property. Section 6.2 is devoted to the proof of Proposition 19. Building upon this result we prove the conservative extension theorem in Section 6.3. Finally, in Section 6.4 we show how these results constitute a theory of implementation type correctness for staged composition synthesis.

6.1 Technical Preliminaries

The following notion of type erasure will play an important role in relating semantic types to implementation types.

**Definition 10** For type expressions \( \theta \in S_i \) we define the erasure of \( \theta \), written \( \theta^o \), \((i = 1, 2)\), as follows.

\[
\begin{align*}
\theta^o & \equiv \theta, \text{ when } \theta \in T_1 \cup T_2 \\
(\theta \cap u)^o & \equiv \theta^o \text{ when } u \in S_1 \cup S_2 \\
(u \cap \theta)^o & \equiv \theta^o \text{ when } u \in S_1 \cup S_2 \\
(\theta \rightarrow \phi)^o & \equiv \theta^o \rightarrow \phi^o \\
(\Theta \cap \phi)^o & \equiv \theta^o \cap \phi^o \\
(\bigcirc \theta)^o & \equiv \bigcirc \theta^o
\end{align*}
\]

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

We shall show below (Lemma 15) that \( \theta_1 = \theta_2 \) implies \( \theta_1^o = \theta_2^o \), hence the operation \( (\cdot)^o \) is a well defined function on equivalence classes with respect to type equality \((=)\). The function \( (\cdot)^o \) erases semantic types in \( S_1 \cup S_2 \) from semantic L1- and L2-types in \( S_1 \cup S_2 \). It is easy to verify that, if \( \theta \in S_i \), then \( \theta^o \in S_i \) and \( \theta^o \) does not contain any subterms from \( S_1 \cup S_2 \). Also, if \( \theta \in T_1 \cup T_2 \) then \( \theta^o \equiv \theta \).

**Lemma 11** For any type substitution \( S \) and \( \theta \in S_1 \cup S_2 \) one has \( S(\theta)^o = S(\theta^o) \).
Proof: By induction with respect to $\vartheta$. □

**Definition 12** A path $\pi$ is a type of the form

$$\pi ::= A \mid \Box \pi \mid \vartheta \to \pi$$

A type $\vartheta$ 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 $\varrho$ is not a function type) is defined to be $n$. We let $\|\vartheta\|$ denote the maximal length of a path in $\vartheta$ (assuming $\vartheta$ is organized).

It is easy to see that any type $\vartheta$ is equal to a polynomially sized organized type $\bar{\vartheta}$, thus, whenever convenient, we shall tacitly assume that types are organized. Every type $\vartheta$ can evidently be written in a (not necessarily unique) standard form as

$$\vartheta = \bigcap_{i \in I} (\vartheta_i \to \varrho_i') \cap \bigcap_{j \in J} \Box \varrho_j \cap \bigcap_{k \in K} A_k$$

The following lemma is a generalization, to include modal types, of a standard lemma [1] for intersection type subtyping (sometimes referred to as “$\beta$-soundness”).

**Lemma 13** Let $\vartheta$ be given in a standard form as

$$\vartheta \equiv \bigcap_{i \in I} (\vartheta_i \to \varrho_i') \cap \bigcap_{j \in J} \Box \varrho_j \cap \bigcap_{k \in K} A_k$$

Then the following conditions hold for all $\varrho, \varrho', A$:

1. $\vartheta \leq \varrho \to \varrho'$ if and only if the set $\{i \in I \mid \vartheta_i \leq \varrho_i\}$ is nonempty and $\bigcap \{\varrho_i' \mid \vartheta_i \leq \varrho_i\} \leq \varrho'$.
2. $\vartheta \leq \Box \varrho$ if and only if $J \neq \emptyset$ and $\bigcap_{j \in J} \varrho_j \leq \varrho$.
3. $\vartheta \leq A$ if and only if $A \equiv A_k$ for some $k \in K$.

Proof: The implications from left to right are proven by induction on the derivation of the subtyping relations. The implications from right to left follow easily by the axioms of subtyping. □

**Lemma 14** For $\vartheta, \varrho \in T_1 \cup T_2$, $\vartheta \leq \varrho$ implies $\vartheta \equiv \varrho$

Proof: By induction with respect to the sum of the sizes of of $\vartheta$ and $\varrho$, using Lemma 13. □

**Lemma 15** For $\vartheta, \varrho \in S_i$ ($i = 1, 2$) $\vartheta \leq \varrho$ implies $\vartheta^o \leq \varrho^o$.

Proof: By induction with respect to the sum of the sizes of $\vartheta$ and $\varrho$, using Lemma 13. □

Notice that Lemma 14 implies that, for $\vartheta, \varrho \in S_i$, we have $\vartheta^o \equiv \varrho^o$ whenever $\vartheta^o, \varrho^o \in T_1 \cup T_2$ and $\vartheta = \varrho$.
6.2 EMBEDDING

The main technical result of this section will be an embedding property (Proposition 19) showing that, under certain conditions, \( \mathcal{C}' \models \varphi \) implies that \( E \) can be transformed to a well typed program in the underlying implementation language by substitution of implementation programs for combinators.

We introduce combinatory expressions over combinator symbols of \( C_1 \) and \( C_2 \), subscripted with types from \( T_1 \) and \( T_2 \), respectively, as follows.

\[
\begin{align*}
f & ::= X_\tau \mid (f_1 f_2) \\
g & ::= F_\sigma \mid (g_1 g_2) \mid \text{box } f
\end{align*}
\]

For a \( C_1 \)-environment \( \mathcal{C} \) in which all bindings are of the form \( (X : \bigcap_{j \in J} \tau_j) \) (intersections of types in \( T_1 \)) and a \( C_2 \)-environment \( \mathcal{D} \) in which all bindings are of the form \( (X : \bigcap_{j \in J} \sigma_j) \) (intersections of types in \( T_2 \)) we define the \( T_1 \)-environment \( \mathcal{C}^+ \) and the \( T_2 \)-environment \( \mathcal{D}^+ \) by

\[
\begin{align*}
\mathcal{C}^+ &= \{(X_\tau_j : \tau_j) \mid j \in J, (X : \bigcap_{j \in J} \tau_j) \in \mathcal{C}\} \\
\mathcal{D}^+ &= \{(F_\sigma_j : \sigma_j) \mid j \in J, (F : \bigcap_{j \in J} \sigma_j) \in \mathcal{D}\}
\end{align*}
\]

We can consider such environments \( \mathcal{C}^+ \) and \( \mathcal{D}^+ \) as \( L_1 \)-environments, resp. \( L_2 \)-environments, by considering (possibly through a mapping, which we shall leave implicit) the symbols \( X_\tau \) (resp. \( F_\sigma \)) as \( L_1 \)-variables (resp. \( L_2 \)-variables).

We define an erasure function \( (\cdot)^- \) mapping these expressions back to combinator expressions of \( C_1 \), respectively \( C_2 \):

\[
\begin{align*}
X_\tau^- &= X \\
(f_1 f_2)^- &= (f_1^- f_2^-) \\
F_\sigma^- &= F \\
g_1 g_2^- &= (g_1^- g_2^-) \\
\text{box } f^- &= \text{box } f^-
\end{align*}
\]

The following definition is needed for our proof of the embedding property (Proposition 19). It generalizes Hindley’s notion of normalization for intersection types in [14] to encompass modal types.

**Definition 16 (Normalization)** We define the normalization of any type \( \theta \), written \( \theta^* \), by

\[
\begin{align*}
A^* &= A \\
(\theta \cap \varphi)^* &= \theta^* \cap \varphi^* \\
(\theta \rightarrow \varphi)^* &= \bigcap_{i \in I} (\theta^* \rightarrow \pi_i), \text{ for } \varphi^* \equiv \bigcap_{i \in I} \pi_i \\
(\Box \theta)^* &= \bigcap_{i \in I} \Box \pi_i, \text{ for } \theta^* \equiv \bigcap_{i \in I} \pi_i
\end{align*}
\]

Types of the form \( \theta^* \) are called normalized.
Normalization is related to organization of types, in that an organized type is normalized at the top level (but not necessarily recursively). Notice that normalization disappears on types in $\mathbb{T}_1 \cup \mathbb{T}_2$, that is, $\theta^* \equiv \theta$ for $\theta \in \mathbb{T}_1 \cup \mathbb{T}_2$. In the proof of Proposition 19 below we shall use the properties stated in the following lemma from [14] (here extended to include the modal constructor).

Lemma 17 (Normalization) One has $\theta^* = \theta$, and if $\theta \equiv \bigcap_{i \in I} \pi_i$ and $\varphi \equiv \bigcap_{i \in J} \pi_i'$ are normalized then one has

$$\theta \leq \varphi \quad \text{if and only if} \quad \forall j \in J. \, \exists i \in I. \, \pi_i \leq \pi_i'$$

The following definition is central for our development towards the conservative extension theorem (Theorem 23). The concepts of supported and grounded types capture relations between semantic types and their “underlying” (under erasure) implementation language types.

Definition 18 (Supported, grounded) For $\theta \in S_i$ ($i = 1, 2$) we say that $\theta$ is supported if and only if $\theta = \bigcap_{i \in I} \pi_i$ with $\pi_i \in \mathbb{T}_i$ for $j \in J$. We say that $\theta$ is grounded if and only if $\theta^* \in \mathbb{T}_i$.

An environment $\mathcal{C}$ or $\mathcal{D}$ is said to be supported (grounded) if all types appearing in the environment are supported (grounded).

We say that a derivation in $C_1$ or $C_2$ is monomorphic, if and only if all applications of rule (var) use the identity substitution (combinatory logics restricted to such derivations are finite combinatory logics, in the sense of [19]). We can now prove the main technical result of this section.

Proposition 19 (Embedding) Let $\mathcal{C}$ and $\mathcal{D}$ be supported. Then

1. $\mathcal{C} \vdash_{e,1} e : \phi$ with $\phi^* \equiv \bigcap_{i \in I} \pi_i$ is derivable by a monomorphic derivation if and only if for all $i \in I$ there are $f_i$ and $\tau_i \in \mathbb{T}_1$ such that $\phi^+ \vdash_{L_1} f_i : \tau_i$ with $(f_i)^- \equiv e$ and $\tau_i \leq \pi_i$.

2. $\mathcal{C} ; \mathcal{D} \vdash_{e,2} E : \psi$ with $\psi^* \equiv \bigcap_{i \in I} \pi_i$ is derivable by a monomorphic derivation if and only if for all $i \in I$ there are $g_i$ and $\sigma_i \in \mathbb{T}_2$ such that $\mathcal{C}^+ ; \mathcal{D}^+ \vdash_{E} g_i : \sigma_i$ with $(g_i)^- \equiv E$ and $\sigma_i \leq \pi_i$.

Proof: To prove the implication from left to right in claim 1, let a monomorphic derivation of $\mathcal{C} \vdash_{e,1} e : \phi$ with $\phi^* \equiv \bigcap_{i \in I} \pi_i$ be given. Since $\mathcal{C}$ is supported, bindings in $\mathcal{C}$ can be written as $(X : \bigcap_{j \in J} \tau_j)$, and we have $\mathcal{C}^+ = \{(X : \tau_j) \mid j \in J, (X : \bigcap_{j \in J} \tau_j) \in \mathcal{C}\}$.

We show that for all $i \in I$ there are $f_i$ and $\tau_i \in \mathbb{T}_1$ with $\mathcal{C}^+ \vdash_{L_1} f_i : \tau_i$ and $\tau_i \leq \pi_i$, by induction with respect to the derivation of $\mathcal{C} \vdash_{e,1} e : \phi$ and by cases over the last rule applied in the derivation.

In case the derivation is by rule (var) of the form $\mathcal{C}, X : \bigcap_{j \in J} \tau_j \vdash_{e,1} X : \bigcap_{j \in J} \tau_j$, we take $f_j \equiv X_{\tau_j}$, and we have $\mathcal{C}^+ \vdash_{L_1} f_j : \tau_j$ by rule (var) and definition of $\mathcal{C}^+$.

Suppose the last rule application of the derivation is an application of rule ($\rightarrow$E) of the form

$$\frac{\mathcal{C} \vdash_{e,1} e' : \phi \rightarrow \phi' \quad \mathcal{C} \vdash_{e,1} e : \phi}{\mathcal{C} \vdash_{e,1} (e'e) : \phi'}$$
where \((\phi \to \phi')^* \equiv \bigcap_{i \in I} \phi^* \to \pi_i^*\) and \(\phi'^* \equiv \bigcap_{i \in I} \pi_i^*\). By induction hypothesis applied to the left premise there are, for all \(i \in I\), \(f_i^*\) with \((f_i^*)^- \equiv e'\) and \(\tau_i^*\) such that
\[
\mathcal{C}^+ \vdash_{\mathbf{u}_i} f_i^* : \tau_i^* \tag{6.1}
\]
and
\[
\tau_i^* \leq \phi^* \to \pi_i^* \tag{6.2}
\]
It follows from (6.2) and Lemma 13 that we must have
\[
\tau_i^* \equiv \tau_i'' \to \tau_i''' \text{ with } \phi^* \leq \tau_i'' \text{ and } \tau_i''' \leq \pi_i^* \tag{6.3}
\]
Let \(\phi^* \equiv \bigcap_{j \in J} \pi_j\). Then we can write \(\phi^* \leq \tau_i''\) from (6.3) as
\[
\bigcap_{j \in J} \pi_j \leq \tau_i'' \tag{6.4}
\]
By induction hypothesis applied to the right premise there are, for all \(j \in J\), \(f_j\) and \(\tau_j\) such that \((f_j^-)^- \equiv e\) and
\[
\mathcal{C}^+ \vdash_{\mathbf{u}_j} f_j : \tau_j \tag{6.5}
\]
and
\[
\tau_j \leq \pi_j \tag{6.6}
\]
Now let \(i \in I\) be given. By the normalization property (Lemma 17) and (6.4) there exists \(j_i \in J\) such that
\[
\pi_{j_i} \leq \tau_i'' \tag{6.7}
\]
From (6.1) and (6.3) we have
\[
\mathcal{C}^+ \vdash_{\mathbf{u}_i} f_i^* : \tau_i'' \to \tau_i''' \tag{6.8}
\]
Taking (6.6) and (6.7) together we have \(\tau_i \leq \pi_{j_i} \leq \tau_i''\), hence \(\tau_{j_i} \leq \tau_i''\). By Lemma 14 the latter implies that, in fact, \(\tau_{j_i} \equiv \tau_i''\). Therefore, by (6.5) we have
\[
\mathcal{C}^+ \vdash_{\mathbf{u}_j} f_j^* : \tau_i'' \tag{6.9}
\]
Hence, by (6.8), (6.9) and rule \((\to E)\) we get
\[
\mathcal{C}^+ \vdash_{\mathbf{u}_i} (f_i^* f_j^*_i) : \tau_i''' \tag{6.10}
\]
The claim now follows from \((f_i^* f_j^*_i)^- \equiv (f_i^-)^- (f_j^-)^- \equiv (e' e)\) and (6.3) \(\tau_i''' \leq \pi_i^*\).
Suppose the last rule application of the derivation is an application of rule $(\land l)$ of the form
\[
\frac{\mathcal{C} \vdash e : \phi}{\mathcal{C} \vdash e : \phi \land \phi'}(\land l)
\]
where $\phi^* \equiv \bigcap_{k \in K} \pi_k$ and $\phi'^* \equiv \bigcap_{j \in J} \pi_j'$. By induction hypothesis applied to the subderivations we have $\mathcal{C}' \vdash f_k : \tau_k$ with $(f_k)^- = e$ and $\tau_k \leq \pi_k$ for all $k \in K$, and $\mathcal{C}^+ \vdash f'_j : \tau'_j$ with $(f'_j)^- = e$ and $\tau'_j \leq \pi'_j$ for all $j \in J$. Since $(\phi \land \phi')^* \equiv \phi^* \land \phi'^*$ the claim follows.

Suppose finally that the last rule application of the derivation is an application of rule $(\leq)$ of the form
\[
\frac{\mathcal{C} \vdash e : \phi}{\mathcal{C} \vdash e : \phi' \leq \phi'}(\leq)
\]
with $\phi^* \equiv \bigcap_{k \in K} \pi_k$ and $\phi'^* \equiv \bigcap_{j \in J} \pi_j'$. By induction hypothesis we have $\mathcal{C}' \vdash f_k : \tau_k$ with $(f_k)^- = e$ and $\tau_k \leq \pi_k$ for all $k \in K$. Since $\phi \leq \phi'$, it follows by Lemma 17 that for any $j \in J$ we can find $k_j \in K$ such that $\pi_{k_j} \leq \pi'_j$. Hence we have $\mathcal{C}' \vdash f_{k_j} : \tau_{k_j}$ with $(f_{k_j})^- = e$ and $\tau_{k_j} \leq \pi'_j$, thereby proving the claim.

To prove the implication from right to left, let $\mathcal{C}' = \{ (X : \bigcap_{j \in J} \tau_j) \mid (X : \tau_j) \in \mathcal{C} \text{ for some } j \in J \}$. Use that $(\mathcal{C}^-) = \mathcal{C}'$, $(f_i)^- = e$ and that $\mathcal{C}^+ \vdash f_i : \tau_i$ together with $\tau_i \leq \pi_i$ for all $i \in I$ clearly implies $(\mathcal{C}^-) \vdash e : \bigcap_{i \in I} \pi_i$.

To prove claim 2 one proceeds as in the proof of claim 1, using claim 1 in applications of rule $(\Box l)$.

\[\square\]

6.3 Conservative Extension

For a derivation tree $D$ in $C_1$ or $C_2$, let $S_D(X)$ (resp. $S_D(F)$) be the set of substitutions $S$ such that $S$ is applied in an application in $D$ of rule (var) of the form $\mathcal{C}', X : \phi \vdash e : X : S(\phi)$ (resp. $\mathcal{C} ; \mathcal{D}, F : \psi \vdash e : F : S(\psi)$). We let $\mathcal{C}^D$, resp. $\mathcal{D}^D$, denote the exponentiated environments defined as follows:
\[
\mathcal{C}^D = \{ (X : \bigcap_{S \in S_D(X)} S(\phi)) \mid (X : \phi) \in \mathcal{C} \}
\]
\[
\mathcal{D}^D = \{ (F : \bigcap_{S \in S_D(F)} S(\psi)) \mid (F : \psi) \in \mathcal{D} \}
\]
We write $\mathcal{C} \vdash^D e : \phi$ whenever $\mathcal{C} \vdash e : \phi$ is derivable by derivation $D$, and similarly for $\mathcal{C} ; \mathcal{D} \vdash^D e : \psi$.

Notice that exponentiation preserves supportedness of environments, since type substitutions preserve supportedness of types: if $\theta \in S_i$ with $\theta = \bigcap_{j \in J} \tau_j$ and $\pi_j \in T_i$, then $S(\theta) = \bigcap_{j \in J} S(\tau_j)$ with $S(\pi_j) \in T_i$.

Let us write $(\cdot)^D = (((\cdot)^\circ)^D)$ and $(\cdot)^{D^+} = (((\cdot)^\circ)^D)^+$. We can reduce derivations in $C_1$ or $C_2$ to monomorphic ones modulo exponentiation of environments.
Lemma 20 In system $C_1$ and $C_2$ one has:

1. $C \vdash^{D} e : \phi$ if and only if $C^{D} \vdash^{e} e : \phi$ by a monomorphic derivation.

2. $C; D \vdash^{D} E : \psi$ if and only if $C^{D}; D \vdash^{e} E : \psi$ by a monomorphic derivation.

Proof: By induction with respect to derivations. □

Proposition 21 Suppose that $C^{o}$ and $D^{o}$ are supported and that $\phi$ and $\psi$ are grounded. Then one has

1. $C^{o} \vdash^{D} e : \phi^{o}$ if and only if $C^{o+D^{+}} \vdash_{t_{1}} f : \phi^{o}$ for some $f$ with $f^{−} \equiv e$.

2. $C^{o}; D^{o} \vdash^{D} E^{o} : \psi^{o}$ if and only if $C^{o+D^{+}}; D^{o+D^{+}} \vdash_{t_{2}} g : \psi^{o}$ for some $g$ with $g^{−} \equiv E$.

Proof: To prove the first claim, use Lemma 20 to see that $C^{o} \vdash^{D} e : \phi^{o}$ if and only if $C^{o+D^{+}} \vdash^{e} e : \phi^{o}$ by a monomorphic derivation. Because $C^{o}$ is supported, $C^{o+D^{+}}$ is supported, and we can apply Proposition 19 to $C^{o+D^{+}} \vdash^{e} e : \phi^{o}$. Since $\phi^{o} \in T_{1}$, hence $(\phi^{o})^{*} \equiv \phi^{o}$, and Proposition 19 therefore shows that $C^{o+D^{+}} \vdash_{t_{1}} f : \tau$ for some $\tau \in T_{1}$ and $f^{−} \equiv e$ and $\tau \leq \phi^{o}$. Since $\phi^{o} \in T_{1}$, it follows from $\tau \leq \phi^{o}$ by Lemma 14 that, in fact, we have $\tau \equiv \phi^{o}$, thereby proving the claim.

The proof of the second claim is analogous. □

Lemma 22 (Erasure)

1. If $C \vdash_{e_{1}} e : \phi$ then $C^{o} \vdash_{e_{1}} e : \phi^{o}$

2. If $C; D \vdash_{e_{2}} E : \psi$ then $C^{o}; D^{o} \vdash_{e_{2}} E : \psi^{o}$

Proof: The proof is by induction in the assumed derivations. Consider the first claim.

In the case the derivation is by rule (var), we have a derivation $C^{′}, X : \phi \vdash_{e_{1}} X : S(\phi)$. Then we have $(C^{′})^{o}; X : \phi^{o} \vdash_{e_{1}} X : S(\phi^{o})$. By Lemma 11, we have $S(\phi^{o}) = S(\phi)^{o}$, which proves the claim.

In the case the last rule applied in the derivation is rule $(\leq)$, we have a derivation of $C \vdash_{e_{1}} e : \phi$ from $C^{′} \vdash_{e_{1}} e : \phi^{′}$ and $\phi^{′} \leq \phi$. By induction, we have $C^{o} \vdash_{e_{1}} e : (\phi^{′})^{o}$. By Lemma 15 we have $(\phi^{′})^{o} \leq \phi^{o}$, and the claim follows by rule $(\leq)$. The remaining cases follow easily from induction hypothesis.

For the second claim, we use the first claim in the case of rule $(\square I)$ together with induction hypothesis. The remaining cases follow by reasoning analogously to the proof of the first claim. □

Theorem 23 (Conservative extension)

1. Suppose that $C^{o}$ and $D^{o}$ are supported and that $\phi$ and $\psi$ are grounded. Then one has
We show that inhabitants obtained by composition in the combinatory systems $\emptyset$ we have $\emptyset$ with bindings in $\emptyset$. By the same reasoning, if $\emptyset; \varnothing \vdash_{\varepsilon} E : \psi$ then $\emptyset; \varnothing \vdash_{\varepsilon} \varnothing; \varnothing \vdash_{\varepsilon} \varnothing : \psi^o$ for some $\psi$ with $\varnothing^o \equiv \emptyset$.

2. Suppose that $\emptyset, \varnothing, \phi$ and $\psi$ are grounded. Then one has

a) If $\emptyset \vdash_{\varepsilon} e : \phi$ is derivable by a monomorphic derivation, then $\emptyset; \varnothing \vdash_{\varepsilon} e : \phi^o$.

b) If $\emptyset; \varnothing \vdash_{\varepsilon} E : \psi$ is derivable by a monomorphic derivation, then $\emptyset; \varnothing; \varnothing \vdash_{\varepsilon} E : \psi^o$.

**Proof:** The first claim follows by first applying Lemma 22, then Proposition 21. The second claim follows by the same properties and noticing, in addition, that the operation $()^D$ vanishes for monomorphic derivations $D$, and that the operation $(\cdot)^{+}$ is superfluous on grounded environments for such derivations.

\[\square\]

### 6.4 Correctness

We show that inhabitants obtained by composition in the combinatory systems $C_1$ and $C_2$ from environments whose erasure represent well typed programs in $L_1$ and $L_2$ can be translated back to well typed expressions of $L_1$ and $L_2$ 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 $L_1$ and $L_2$, respectively, such that

\[
\begin{align*}
\emptyset; \emptyset \vdash_{\varepsilon} T_X : \tau_X & \text{ for } X \in \mathcal{X} \\
\emptyset; \emptyset \vdash_{\varepsilon} M_F : \sigma_F & \text{ for } F \in \mathcal{F}
\end{align*}
\]

and exposed in combinatory environments as

\[
\begin{align*}
\emptyset & = \{(X : \phi_X) \mid X \in \mathcal{X}\} \\
\varnothing & = \{(F : \psi_F) \mid F \in \mathcal{F}\}
\end{align*}
\]

with $(\phi_X)^o = \tau_X$ and $(\psi_F)^o = \sigma_F$. Then $\emptyset$ and $\varnothing$ are grounded, hence $\emptyset^o$ and $\varnothing^o$ are supported and $\emptyset^o D^+$ and $\varnothing^o D^+$ are grounded.

Suppose now that $\emptyset \vdash_{\varepsilon} e : \phi$ with $\phi$ grounded. It follows from Theorem 23 that we have $\emptyset^o D^+ \vdash_{\varepsilon} f : \phi^o$ for some $f$ with $f^o \equiv e$. Since all combinators in $\emptyset^o D^+$ have the form $(X_S : S(\tau_X))$ with $(X : \phi_X) \in \emptyset$ and $(\phi_X)^o \equiv \tau_X$, it follows by Lemma 1 and Lemma 2 that we have $\emptyset; \emptyset \vdash_{\varepsilon} f' : \phi^o$ where

\[f' \equiv f[X_S(\tau_X) := S(T_X)]\]

By the same reasoning, if $\emptyset; \varnothing \vdash_{\varepsilon} E : \psi$ with $\psi$ grounded, we have $\emptyset^o D^+; D^o D^+ \vdash_{\varepsilon} g : \psi^o$ with bindings in $\emptyset^o D^+$ all of the form $(F_S : S(\sigma_F))$ with $(F : \psi_F) \in \varnothing$ and $(\psi_F)^o \equiv \sigma_F$. Hence, by Lemma 3 and Lemma 4 we have $\emptyset; \emptyset \vdash_{\varepsilon} g' : \psi^o$ where

\[g' \equiv g[X_S(\tau_X) := S(T_X)][F_S(\sigma_F) := S(M_F)]\]

32
We provide a theoretical (semi-) decision procedure for solving the relativized inhabitation problems for $C_1$ and $C_2$, which, as will be explained, is a decision procedure for bounded variants of the inhabitation problem. The procedure underlies the optimized implementation in the (CL)S system discussed in Chapter 8.

The following path lemmas can be proven using the techniques employed in [10] to prove the corresponding path lemma for bounded combinatory logic. The proofs are by induction in the parameter $m$, using Lemma 13. Since it is straightforward to transfer the proofs from [10] they are left out. Based on the path lemmas we can specify a semi-decision procedure for the inhabitation problem $C; D \vdash C_2 : \psi$, as shown in Figure 7.1, by the results of [10].

For a type $\vartheta \equiv \vartheta_1 \to \cdots \to \vartheta_n \to \varrho$ we let $\text{arg}_i(\vartheta) \equiv \vartheta_i$ for $1 \leq i \leq n$, and we let $\text{tgt}_m(\vartheta) \equiv \varrho$. Finally, we let $P_m(\vartheta)$ denote the set of paths in $\vartheta$ of length at least $m$.

**Lemma 24 (Path Lemma for $C_1$)** The following are equivalent conditions:

1. $C \vdash_{c_1} X e_1 \ldots e_m : \psi$;
2. There exists a finite set of substitutions $S \subseteq V \to T_0 \cup S_1 \cup S_2$ and a set $P$ of paths in $P_m(\bigcap \{S(C(X)) \mid S \in S\})$ such that
   a) $\bigcap_{\pi \in P} \text{tgt}_m(\pi) \leq \psi$;
   b) $C \vdash_{c_1} e_i : \bigcap_{\pi \in P} \text{arg}_i(\pi)$, for all $i \leq m$.

**Lemma 25 (Path Lemma for $C_2$)** The following are equivalent conditions:

1. $C; D \vdash_{c_2} E : \psi$;
2. Either $\psi = \Box \varphi$, $E \equiv \text{box } e$ and $C \vdash_{c_1} e : \varphi$, or $E \equiv F E_1 \ldots E_m$ and there exists a finite set of substitutions $S \subseteq V \to T_0 \cup S_1 \cup S_2$ and a set $P$ of paths in $P_m(\bigcap \{S(D(F)) \mid S \in S\})$ such that
   a) $\bigcap_{\pi \in P} \text{tgt}_m(\pi) \leq \psi$;
   b) $C; D \vdash_{c_2} E_i : \bigcap_{\pi \in P} \text{arg}_i(\pi)$, for all $i \leq m$.

By restriction to monomorphic derivations [19] or by bounding the size of substitutions to depth $k$ in derivations [10] the semi-decision procedure shown in Figure 7.1 becomes a decision procedure (see Theorem 26 below). We use the notation of [10] where $\text{choose}$ denotes non-deterministic choice and $\text{forall}$ denotes universal branching of an alternating Turing-machine [2].
Input: \( \mathcal{C}, \mathcal{D}, \vartheta \)

loop:

1. if \( (\vartheta \in S_1) \) then
2. choose \( (X : \phi) \in \mathcal{C} \)
3. choose \( S \subseteq \mathcal{V} \rightarrow T_0 \cup \mathcal{G}_1 \cup \mathcal{G}_2 \)
4. \( \psi' := \bigcap \{ S(\phi) \mid S \in S \} \)
5. choose \( m \in \{0, \ldots, ||\phi'||\} \)
6. choose \( P \subseteq P_m(\psi') \)

7. if \( (\bigcap_{\pi \in P} \mathfrak{t}_m(\pi) \leq \vartheta) \) then
8. if \( (m = 0) \) then accept;
9. else
10. for all \( (i = 1 \ldots m) \)
11. \( \vartheta := \bigcap_{\pi \in P} \mathfrak{a}_i(\pi) \)
12. goto loop;
13. else goto case1 or goto case2
14. end

15. if \( (\vartheta = \square \phi) \) then
16. goto case1 or goto case2
17. else goto case2
18. case1:
19. \( \vartheta := \phi \); goto loop
20. case2:
21. choose \( (F : \psi) \in \mathcal{D} \)
22. choose \( S \subseteq \mathcal{V} \rightarrow T_0 \cup \mathcal{G}_1 \cup \mathcal{G}_2 \)
23. \( \psi' := \bigcap \{ S(\phi) \mid S \in S \} \)
24. choose \( m \in \{0, \ldots, ||\psi'||\} \)
25. choose \( P \subseteq P_m(\psi') \)

26. if \( (\bigcap_{\pi \in P} \mathfrak{t}_m(\pi) \leq \vartheta) \) then
27. if \( (m = 0) \) then accept;
28. else
29. for all \( (i = 1 \ldots m) \)
30. \( \vartheta := \bigcap_{\pi \in P} \mathfrak{a}_i(\pi) \)
31. goto loop;

Figure 7.1: Alternating Turing machine semi-decision procedure for \( \mathcal{C}; \mathcal{D} \vdash \vartheta \) : \( \vartheta \)

7.1 Expressiveness and Complexity

Perhaps surprisingly, the restriction to grounded types in Theorem 23 does not limit the theoretical expressive power of the inhabitation relation, even though the theorem shows (see
Chapter 6.4) that restricted inhabitation in C1 and C2 implies inhabitation in the systems restricted to implementation language types of L1 and L2 which are simply typed (and this would remain so even in case L1 has only constant types).\footnote{As shown in [19], monomorphic (finite) inhabitation simple types is in PTIME, whereas it is EXPTIME-complete with intersection types; in the bounded case [10], inhabitation is EXPTIME-complete in simple types independently of the bound, whereas it is \((k + 2)\)-EXPTIME-complete with bound \(k\) in the presence of intersection types.}

**Theorem 26 (Complexity)** Inhabitation in the systems C1 and C2 is \((k + 2)\)-EXPTIME-complete with bound \(k\) (as in [10]) and EXPTIME-complete in the monomorphic case (as in [19]).

**Proof:** The hardness proof is implicit in the techniques used in the hardness proof in [10]. There, it is shown how to encode alternating space bounded Turing machines using a bounded version of system C1. In the encodings, simple types generated from a single constant type, called “\(\bullet\)” there, are employed as components in intersection types, to regiment the types in the constructions. It is easy to verify that these encodings are all carried out with inhabitants that are restricted to be typable in simple types built from \(\bullet\) when all other types are erased (corresponding to applying our erasure function \((\gamma)\)). Thus, the types used for the combinators in the lower bound construction of [10] are all grounded in the sense of Theorem 23.

The intuitive reason behind this result can be explained by the fact that the combinators required to encode the Turing machines in the lower bound proofs have simple monomorphic types that allow them to be iterated and combined in sufficiently arbitrary ways, and the hardness of the inhabitation problem arises from intersection types being used to restrict to those combinations that simulate accepting runs of the machine. In other words, restricting to the set of inhabitants \(\{e \mid e\text{ is simply typed}\}\) yields no theoretic simplification. And asking for an inhabitant in that set which type checks with intersection types still requires search in that set for a member typable in intersection types.

Not notwithstanding these theoretical results, it remains an extremely intriguing opportunity for optimizations in practice that we could exploit the simpler type structure (without semantic intersection types) of implementation language types (see Chapter 8).
We implemented the presented framework in the context of the (CL)S (Combinatory Logic Synthesizer) tool\(^1\) \([8, 18]\), using F# and C#. The inhabitation algorithm is configured for the bound \(k = 0\), limiting type instantiations to atomic types or intersections of such (the theoretical complexity of inhabitation is 2-\text{Exptime}-complete for this bound, see Chapter \(7\)). We implemented an interpreter for L2 \((\lambda \rightarrow \rightarrow \ast)\) of \([4]\) under the semantics of \(\rightarrow \rightarrow \#\). We used this interpreter to obtain L1-programs from inhabitants generated by solving the inhabitation problem in (CL)S. The algorithm has been optimized and parallelized, making it suitable for multi-core environments and compute-clusters. In addition, (CL)S provides various features including extended logging, and various graphical representations of the execution of the inhabitation algorithm by means of execution graphs.\(^2\)

In this section 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. In particular, it is not possible to present the L2-implementations of all \(\varnothing\)-combinators or the generated L1-code, confining discussion to a few interesting combinators.

We first extended the L1-repository \(C\) introduced in Chapter 5 to a full version of the tracking-scenario consisting of 8 combinators (discussed in \([18, \text{Figure 8}]\)), allowing to project a tracked object to its coordinates, for example. Furthermore, we added the following L2-combinators with associated implementations to \(\varnothing\):\(^3\)

\[
\begin{align*}
\text{avgFun} &: \square (\text{TrObj} \rightarrow R \cap a \cap ms) \rightarrow \square (\text{TrObj}) \rightarrow \square (R \cap a \cap \text{Avg} \cap ms) \\
\text{dist} &: \square ((R, R) \cap \text{Cart}) \rightarrow \square ((R, R) \cap \text{Cart}) \rightarrow \square (R \cap \text{dist})
\end{align*}
\]

Here, \text{avgFun} uses code of a function that extracts a measured real value (with semantic property \(a\)) from a tracked object and code of an array of such objects to produce code of a value that is an average. Similarly, \text{dist} calculates distances between two coordinates. We conducted experiments with these repositories synthesizing code for functions that cal-

---

\(1\) \url{http://www-soof.cs.tu-dortmund.de/seal/cls_de.shtml}

\(2\) We refer to the first section of the appendix for a more detailed description of (CL)S and our experiments including repositories and input files to the synthesis algorithm, generated code, and execution times as well as screenshots of the generated execution graphs. In our examples and in our implementation we extend L1 with data type constructors and built-in functions. We limit distributivity of type constructors of arity greater than 1 over intersection, since full distributivity requires extensions to our theory and is expensive for inhabitation while providing no big advantage in practice.

\(3\) [\(\varnothing\)] is a unary type constructor representing an array of objects of type \(\varnothing\).
culate various relevant values for tracked objects. Using (CL)S we solved various inhabitation questions for this scenario, e.g., \( \mathcal{C} : S \vdash_{C2} \) : \( \square (R \cap \text{dist}) \). The resulting inhabitant, \( \text{dist} (\text{boxcdn} (\text{pos} (\text{TrV}(0))), \text{boxcdn} (\text{pos} (\text{TrV}(0)))) \), has L1-code of type real describing a distance between objects. All synthesis-requests of this form were answered in \( \leq 250 \) ms.

In the second example scenario we consider repositories \( \mathcal{C} \) and \( \mathcal{D} \) for synthesizing sorting routines for arrays of objects for which an order relation is given. First, assume that \( \mathcal{C} \) only contains an L1-combinator \( \text{lessThan} : ((R, R) \to \text{bool}) \cap \text{incTO} \) deciding the standard total order \( \leq_R \) on reals, where \( \text{incTO} \) is a semantic type stating that it is an increasing total order. The repository \( \mathcal{D} \) contains the combinators with associated implementations:

\[
S : \square((\alpha, \beta) \to \text{bool}) \to \square((\bar{\alpha} \to \bar{\alpha})) \cap (\square a \to \square(\top \to a \cap \text{Sorted})) \\
\text{swap} : (\square((\bar{\alpha}, \beta) \to \gamma)) \to \square((\bar{\beta}, \bar{\alpha}) \to \gamma)) \cap (\square a \to \square \text{Rev}(a)) \\
\Phi : (\square((\bar{\alpha}, \bar{\alpha}) \to \text{bool}) \to \square((\bar{\alpha}, \bar{\alpha}) \to \text{bool})) \cap \\
(\square \text{Rev}(\text{incTO}) \to \square \text{decTO}) \cap (\square \text{Rev}(\text{decTO}) \to \square \text{incTO})
\]

The combinator \( S \) contains an L1-template that realizes bubble sort. Its first type component states that given L1-code of a binary relation, \( S \) produces L1-code of 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 (\( \text{decTO} \)), then \( S \) returns code of a function that sorts a typed array in decreasing order. The combinator \( \text{swap} \) is the modal version of the \( \lambda \)-calculus combinator that swaps the arguments of a function with a semantic property expressing that it reverses the order of arguments (\( \text{Rev} \)). The type of the combinator \( \Phi \) (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 \( \mathcal{C} \vdash_{C2} \) : \( \square (R \to (R \cap \text{decTO} \cap \text{Sorted})) \) produces the following inhabitant:\footnote{Note that our tool can be configured to only consider acyclic inhabitants.} \( S(\Phi\text{swap}(\text{boxlessThan})) \). Its \( \text{\textbullet} \to^* \)-reduction in L2 results in a corresponding L1-sorting routine as described above. The subterm \( \Phi\text{swap}(\text{boxlessThan}) \) creates an L1-function of the form \( \text{fn}(x, y) : (R, R) \Rightarrow \text{lessThan}(y, x) \). In the semantic universe, 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 \( \text{\textbullet} \to^* \)-reduction produced an implementation of bubble sort for sorting an array of reals in decreasing order (24 lines of L1-code) within 150 ms. Changing the inhabitation question by replacing \( \text{decTO} \) by \( \text{incTO} \) the algorithm used \( \text{lessThan} \) to produce a bubble sort implementation for sorting reals in increasing order. We extended the repositories in various ways, e.g., adding a combinator producing a partial order on the nodes of a directed acyclic graph from the graph’s edge-relation can be used to synthesize a sorting routine for topologically sorting the nodes.

Our last example combines the two scenarios presented above. The example 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 (box tmp) • (box Tr) described in Chapter 5. This function is 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 filter function that filters out all objects from the array whose temperature is below average. Finally, the remaining array is sorted in decreasing order (with regard to the temperature) such that the object with the highest temperature is the first entry of the array.\(^5\) We assume that \(\mathcal{D}_3\) contains the combinator \(\text{filterAndSort}\) which will be the top-level combinator for the inhabitant realizing the desired function. Amongst others \(\text{filterAndSort}\) requires an argument whose type is given by the following combinator:

\[
\text{largerThanAvg} : \Box(\Box \alpha \to \beta \cap \text{Cel} \cap \text{Avg}) \to \Box(((\beta, \beta) \to \text{bool}) \cap \text{decTO}) \to \\
\Box[\alpha] \to \Box(\alpha \to \beta \cap \text{Cel}) \to \Box(\alpha \to \text{bool})
\]

The combinator \(\text{largerThanAvg}\) requires code of a function which calculates an average temperature of an array of objects of type \(\alpha\), code of a decreasing total order, code of an array of objects of type \(\alpha\), and code of a function that returns the temperature of an object of type \(\alpha\). Given these types, it first calculates the average temperature of the objects in the given array by using the function provided as a first argument. Then it uses the total order and the function calculating the temperature to compare the temperature of an object of type \(\alpha\) to the calculated average temperature, returning true if it is larger than this average. By posing the inhabitation question \(\mathcal{S}_5; \mathcal{D}_3 \vdash C_2? : \Box(\Box \alpha \to \Box \alpha \cap \text{decTO} \cap \text{Sorted})\) in \((\text{CL})\mathcal{S}\) we obtained the following inhabitant:

\[
\text{filterAndSort}((\text{box tmp}) \cdot (\text{box TrV}), \Phi(\text{swap}(\text{box lessThan})), \text{avgFun}, \text{largerThanAvg}, F)
\]

As can be seen, \(\text{largerThanAvg}\) is used as an argument for \(\text{filterAndSort}\) with an interesting interaction between these two combinators, in that the L2-implementation of \(\text{filterAndSort}\) uses its other arguments to compute code with types for the arguments required by \(\text{largerThanAvg}\). The L2-implementation of \(\text{filterAndSort}\) binds this code to names which are then passed to \(\text{largerThanAvg}\). Thus, \(\text{largerThanAvg}\) may indirectly use functionality created by higher-order applications occurring in \(\text{filterAndSort}\) even though \(\text{largerThanAvg}\) exists outside \(\text{filterAndSort}\). This is possible because \(\text{filterAndSort}\) takes \(\text{largerThanAvg}\) as argument in the first place and can thus provide it with bindings. The time required to synthesize this combinatory term and to reduce it with regard to \(\to^*\) was approximately 9 s. The resulting L1-program realizing the desired functionality has 61 lines of code.\(^6\)

\(^5\) Such a program may very well be useful in a scenario where the tracked objects represent reefer containers and it is necessary to take action on those containers first that are the furthest above average. We discuss two combinators in detail that illustrate the various features mentioned above. The complete repositories can be found in Chapter C.

\(^6\) The details of this example, in particular the repositories, the implementations, and the generated code can be found in Chapter C.
We conclude the discussion of (CL)S by mentioning an important principle for optimization of the inhabitation algorithm. It can be seen from line 5 of the alternating Turing machine presented in Figure 7.1, that the complexity arises from the construction of all possible type substitutions. Thus, one important driver for optimization is to reduce the number of substitutions that actually have to be constructed as far as possible. Using the conservativity property (Theorem 23) and that inhabitants must be well typed in the implementation languages (Chapter 6.4) the number of relevant type substitutions can be drastically decreased heuristically. This principle was used to optimize the inhabitation algorithm of (CL)S and was observed to have major impact on runtime. Our initial experiments with staged composition synthesis are encouraging, but further experiments and engineering are needed, and we are far from having exhausted the optimization potential. Interestingly, the bounded problem for $k = 0$ is comparable to other synthesis problems (e.g., LTL synthesis) complete in 2-EXPTIME but appears to be algorithmically quite different. We expect that future work will raise many questions concerning engineering Hilbert-style propositional logics which is not as well understood as it should be.
The idea of a staged approach to component-oriented synthesis does not appear to have been considered before. Our development of staged composition synthesis would not, however, have been possible without the benefit of the modal analysis by Davies and Pfenning [4] of staged computation and their calculus $\lambda_e^\square \rightarrow$. Not only can we transfer results from $\lambda_e^\square \rightarrow$ 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 [15] with intersection types [1] was introduced and developed in [19, 10, 8, 18, 9]. The complexity theoretical and algorithmic foundations of the relativized inhabitation problem were laid down in [19, 10] (EXPTIME-completeness for the monomorphic restriction, $(k + 2)$-EXPTIME-completeness for the $k$-bounded restrictions). The lower-bound techniques provide insights into the expressive power of inhabitation in the combinatory framework, including connections to tree automata [19] and alternating Turing-machines [10]. The (CL)S-tool has been under development since 2011 and has been applied in several application scenarios, including GUI-generation and the generation of control programs for LEGO NXT robots [8]. Several optimizations have been implemented in the tool, including optimizations based on DFS-look-ahead strategies with subtype matching [9].

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) [16]. Also, our approach can be broadly compared in spirit (rather than in technology) to synthesis of loop free programs [12]. 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. The specification language, the logic and the algorithmics of combinatory logic synthesis are distinct in being based on type theoretical and proof theoretical ideas for propositional logics of the Hilbert type. A general introduction to the combinatory standpoint can be found in [18].

Our approach is related to adaptation synthesis via proof counting [13, 20], where semantic types are combined with proof search in a specialized proof system. In particular, we follow [13, 20] in using semantic specifications at the interface level. The idea of adaptation synthesis [13] 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 [13] the specification language used is a typed predicate logic). Semantic intersection types can be compared to refinement types [11], but semantic types are more general in that they do not need to stand in a refinement relation to implementation types (it can be seen from our
examples that requiring refinement would be a nonacceptable restriction for our applications in synthesis). Still, refinement types are a great source of inspiration for how semantic types can be used in specifications in many interesting situations.
CONCLUSION

We have introduced a framework for staged composition synthesis 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 staged composition synthesis. Further work includes optimizations of the algorithm (in particular, such that further exploit the conservative extension property), more experimentation and applications.
In the appendix we describe some features of (CL)S, including screenshots of the tool and of example input repositories. Furthermore, we present a more detailed discussion of the examples in Chapter 8. In particular, we give the complete repositories from the examples, their implementation in the native language respectively in the metalanguage, and the synthesized L1-code. We also state some figures concerning execution time that (CL)S required for the synthesis in the examples.
We implemented the algorithm presented in Chapter 7 in F# and C#, using the .NET-framework. Here, the core inhabitation procedure is implemented in F# and the control algorithm scheduling the calls to the inhabitation procedure for the various inhabitation tasks is implemented in C#. We enriched the algorithmic framework in various ways in order to increase usability and performance of the combinatory logic synthesis, thus resulting in the tool (CL)S. (CL)S is provided as a (web-)service with exposed endpoints offering SOAP and REST access. Furthermore, the implementation is parallelized. This allows for a usage in our cluster computing environment with multiple concurrent users. Also note that we extended (CL)S by various logging functionalities that are crucial with regard to debugging and analysing experiments. We discuss its features in the following. Concrete examples of type repositories, implementation repositories and synthesized code can be found in Chapter C.

B.1 Type Language

We implemented a type language for intersection types. The internal representation of these types is rather cumbersome, for example, a an arrow type \( a \rightarrow b \) cannot be written in infix notation but has the form \( \text{Arrow}(\text{Var}(a),\text{Var}(b)) \). This makes more complex types non human-readable, which would result in a synthesis-tool with hardly any usability at all. Therefore, we hid the internal structure from the user, by defining an input language that is very close to the actual mathematical definition of intersection types. In this input language constants are given by strings, an arrow type can be written using the right-associative infix symbol \(-\rightarrow\), intersections are represented as lists \([..]\) of its corresponding components, and the type constructor \(\square\) is represented by \#. Here are some examples:

\[
a \rightarrow b \equiv a \rightarrow b \\
(a \rightarrow b \rightarrow c \cap d \equiv a \rightarrow b \rightarrow c,d]
\]

\[(\square(a \cap (b \rightarrow c) \rightarrow d \rightarrow e) \rightarrow \square d) \cap \square a \equiv [\#[a,(b\rightarrow c)\rightarrow d\rightarrow e]\rightarrow \#d,\#a]
\]

We furthermore extended the type-language by covariant constructors of arbitrary arity. Such a constructor is written as

\[
\text{name}_1(_\ldots_,_\ldots_)
\]

where the argument positions can be filled with arbitrary types. Thus, for example, we can define types of the form

\[
[\text{Pair(int,int)}\rightarrow \text{Pair(real,real)},\text{embedding}]
\]
which represents the type of an embedding of $\mathbb{Z}^2$ to $\mathbb{R}^2$.

Partial orders on atomic types can be defined as follows:

\[ \text{int} \leq \text{real} \]

Finally, type variables are written in the form \text{name.varKind} where \text{name} can be any string and \text{varKind} describes the range of atoms that the variable may be mapped to. We only realized level-0 substitutions, meaning that any variable may only be mapped to an atom (atomic substitutions) or an intersection of atoms (Level-0 substitutions). Even these seemingly simple type substitutions already allow for great expressiveness as can be seen by the corresponding complexity results for inhabitation. Atomic substitutions are defined as follows

\[ \text{varTemp} \sim > \text{Celsius, Fahrenheit}, \]

Level-0 substitutions are defined as follows

\[ \text{varFunctionProperty} => \text{surjective, injective, increasing, decreasing}. \]

The first variable kind states that any variable \text{alpha.varTemp} can either represent a temperature in Celsius or in Fahrenheit. Any variable of kind \text{varFunctionProperty} can be instantiated with an intersection of any subset of types in the range. For example, \text{beta.varFunctionProperty} may be mapped to \{surjective, injective, decreasing\}.

### B.2 Inhabitation

Inhabitation requests can be initiated by specifying type repositories \( C \) and \( D \) (represented as \text{combinatorsC} and \text{combinatorsD} in (CL)S) and running the inhabitation algorithm on an input of the form:

\[
\begin{align*}
\text{base} & : // \text{native type constants} \\
\text{D1} & : // \text{semantic L1-type constants} \\
\text{D2} & : // \text{semantic L2-type constants} \\
\text{kinds} & : // \text{varKind-definitions} \\
\text{subtypes} & // \text{subtype definitions} \\
\text{combinatorsC} & // \text{type assumptions in C} \\
\text{combinatorsD} & // \text{type assumptions in D} \\
\vdash ? : \tau
\end{align*}
\]
Corresponding inhabitants are represented as so called AppTerms that are represented as follows

\[ \text{name}<\text{type substitution}>\rightarrow \text{(AppTerm,...,AppTerm)} \]

where the type substitution and the arguments are optional.

Note that we enhanced our editor with code-highlighting for our type-language as depicted in Figure B.1.

---

B.3 L2-repositories

For every combinator in combinatorD we represent a corresponding implementation entry. These entries are written as:

\[ \text{combinatorName} : \{ M \} \]

Here, \( M \) represents the implementation of \( \text{combinatorName} \) in L2 and is defined by:

\[
M ::= \text{box}\ [\text{ListOfL1Code}] \mid \text{let box} \ u = \{ M \} \ \text{in} \ \{ M \} \ | \ x \ | \lambda \ x \ . \ \{ M \} \ | \ M \ M
\]

Here, \( u \) represents a metavariable of L1 and ListOfL1Code is a list of strings and metavariables of L1 (note that we did not implement the syntax of L1, instead we represent code as strings).

For the syntax of \( M \) above we implemented the \( \rightarrow^{\beta} \)-reduction defined in Chapter 2 which resolves all \( \beta \)- and box-reduces. Applying this reduction to an inhabitant will first replace all
combinator names by their corresponding implementation and then result in L1-code (represented as a string, as mentioned above).

Note that we further enhanced our editor with code-highlighting for L2 as depicted in Figure B.2.

![Figure B.2: Code-highlighting for L2](image)

### B.4 Graphical Representation of Inhabitation Requests

Going into the details of the inhabitation algorithm presented in Chapter 7 it can be seen that an inhabitation question may spawn a number of new inhabitation questions which have to be solved in order for the original question to be solved. Each of these inhabitation questions is treated as a separate task where the overall execution structure can be depicted as an execution graph that represents the connection between the various inhabitation tasks. The graph has nodes that represent the instantiated inhabitation tasks and nodes that represent combinators that were chosen to try and solve the respective inhabitation tasks.

We extended (CL)S by a functionality to display these execution graphs in Microsoft Visual Studio 2012. Figure B.3 depicts this editor extension (cf. Figure C.1 for an execution graph of a more complex inhabitation question).
The red root node represents the original inhabitation question and blue nodes represent instantiated inhabitation questions. An orange node that is a child of a node representing an inhabitation question represents a combinator that can be used to inhabit the corresponding target type if provided with suitable arguments. A blue node that is a child of an orange node represents the types of these arguments. A green child of a node representing an inhabitation question represents a combinator that can be used to inhabit the corresponding target type without further arguments. The nodes have been furnished with various kinds of data (e.g., the required computation time per node) that were relevant for the execution of the algorithm. These graphs were particularly helpful during the development of the inhabitation algorithm in order to find errors in our implementation. Furthermore, they are very interesting with regard to analysis of inhabitation questions and the resulting inhabitants. We also included a graphical representation of the inhabitants as term-graphs into (CL)S (cf. Figure B.4).
Figure B.4: Term graph
In this section we present the three example-repositories discussed in Chapter 8, their implementations and in particular the resulting L1-code. We also present figures regarding execution time.

C.1 TRACKING SCENARIO

This section presents the details of the tracking scenario discussed in Chapter 8.

C.1.1 Type Repository

The combinators are typed as follows:

\[
\begin{align*}
\text{base} & : \text{TrObj}, R \\
D1 & : \text{Cart}, \text{Gpst}, \text{Cel}, Fh, \text{ms}, \text{Avg}, \text{UTC}, \text{Pos}, \\
& \quad \text{Coord}, Cx, \text{Polar}, \text{Radius}, Cy, \text{Angle}, \text{Distance} \\
D2 & : \text{Conv} \\
\text{kinds} & \\
\text{varReal} & \Rightarrow R \\
\text{varTemp} & \Rightarrow \text{Cel}, Fh \\
\text{varProperty} & \Rightarrow \text{ms}, \text{Avg}, \text{Pos} \\
\text{varCoord} & \Rightarrow \text{Cart}, \text{Polar} \\
\text{varTime} & \Rightarrow \text{Gpst}, \text{UTC} \\
\text{subtypes} & \\
\text{Cart} & \subseteq \text{Coord}, \text{Polar} \subseteq \text{Coord} \\
\text{combinatorsC} & \\
O & : \text{TrObj} \\
\text{Obj1 Obj2} & : P(\text{TrObj}, \text{TrObj}) \\
V & : \text{ArrayList}(\text{TrObj}) \\
\text{TrV} & : \text{TrObj} \rightarrow D([P(R,R), \text{Cart}], [R, \text{Gpst}], [R, \text{Cel}]) \\
\text{tmp} & : D([P(R,R), R, [R, \text{a.varTemp}]], [R, \text{a.varTemp}, \text{ms}]) \\
\text{pos} & : D([P(R,R), \text{eps.varCoord}], [R, \text{eta.varTime}], R) \rightarrow \\
& \quad [P(P(R,R), \text{eps.varCoord}], [R, \text{eta.varTime}], \text{Pos}] \\
\text{cdn} & : [P(P(R,R), \text{eps.varCoord}], \text{Pos}] \rightarrow [P(R,R), \text{eps.varCoord}] \\
\text{fst} & : [P(R,R), \text{Coord}] \rightarrow R, \text{Cart} \rightarrow Cx, \text{Polar} \rightarrow \text{Radius} \\
\text{snd} & : [P(R,R), \text{Coord}] \rightarrow R, \text{Cart} \rightarrow Cy, \text{Polar} \rightarrow \text{Angle} \\
\text{builtIn2pl} & : [P(R,R), \text{Cart}] \rightarrow [P(R,R), \text{Polar}] \\
\end{align*}
\]
combinatorsD

Composition : \( (\text{TrObj} \rightarrow \{\text{R}, \text{a.varTemp}, \text{ms}\}) \rightarrow \{\text{R}, \text{b.varTemp}, \text{ms}\} \rightarrow \{\text{R}, \text{gamma.varReal}, \text{ms}\}, \text{Conv} \) \rightarrow \{\text{R}, \text{b.varTemp}, \text{ms}\} \)

\( \text{TrObjToCoordinate} : \{\text{TrObj} \rightarrow \{\text{P}(\text{R}, \text{R}), \text{eps.varCoord}, \{\text{R}, \text{eta.varTime}\}, \text{R}\} \) \rightarrow
\( \{\text{P}(\text{P}(\text{R}, \text{R}), \text{eps.varCoord}, \{\text{R}, \text{eta.varTime}\}, \text{R}), \text{Pos}\} \rightarrow \{\text{P}(\text{R}, \text{R}), \text{eps.varCoord}\} \rightarrow \{\text{TrObj} \rightarrow \{\text{P}(\text{R}, \text{R}), \text{eps.varCoord}\}\}

\( \text{Bullet} : \{\alpha.varTemp, \gamma.varReal, \text{ms}\} \rightarrow \{\alpha.varTemp, \gamma.varReal\} \rightarrow \{\beta.varTemp, \gamma.varReal\}, \text{Conv} \rightarrow \{\beta.varTemp, \gamma.varReal, \text{ms}\} \)

\( \text{cl2fh} : \{\text{R}, \text{Cel}, \text{a.varProperty}\} \rightarrow \{\text{R}, \text{Fh}, \text{a.varProperty}, \text{Conv}\} \)

\( \text{Diamond} : \{\alpha.varTemp, \gamma.varReal, \text{ms}\} \rightarrow \{\alpha.varTemp, \gamma.varReal\} \rightarrow \{\beta.varTemp, \gamma.varReal\}, \text{Conv} \rightarrow \{\beta.varTemp, \gamma.varReal, \text{ms}\} \)

\( \text{Avg} : \{\text{TrObj} \rightarrow \{\text{R}, \gamma.varTemp, \text{ms}\} \rightarrow \{\text{R}, \gamma.varTemp, \text{Avg}, \text{ms}\} \rightarrow \{\text{R}, \gamma.varTemp, \text{Avg}, \text{ms}\} \)

\( \text{mapply} : \{\alpha.varReal, \text{Cel}, \gamma.varProperty\} \rightarrow \{\alpha.varReal, \text{Fh}, \gamma.varProperty\} \rightarrow \{\alpha.varReal, \text{Fh}, \gamma.varProperty\} \rightarrow \{\text{TrObj} \rightarrow \{\alpha.varReal, \text{Cel}, \gamma.varProperty\}\rightarrow \{\text{TrObj} \rightarrow \{\alpha.varReal, \text{Cel}, \gamma.varProperty\}\}

\( \text{distance} : \{\text{P}(\text{R}, \text{R}), \text{Cart}\} \rightarrow \{\text{P}(\text{R}, \text{R}), \text{Cart}\} \rightarrow \{\text{R}, \text{Distance}\} \)

\( |-? : \{\text{R}, \text{Distance}\} \)

\textbf{c.1.2 L2-Implementations}

The L2-Implementations of the combinators are as follows:

declarations

\( \text{Bullet} : \{\text{lambda G, }\}
\{\text{lambda F, }\}
\{\text{letbox f = \{ F \} in}
\{\text{letbox g = \{ G \} in}
\{\text{box ['fn ' <y_declarations> ' : TrObj =>}
\{' g ' (" f " \text{\pi <y_declarations> "}))"]}))\}

54
cλ2h : { λz.  
        {let u = {z} in  
         {box [{"let x : R = " u " in x * (9 div 5)+32"]}}}}

Diamond : {λz.  
         {λF. {F z}}}

A : {λH.  
        {λv.  
         {let x h = {H} in  
          {let x u = {v} in  
           {box [{"let " " : ref in t = ref 0;  
                   let " <sum> " : ref R = ref 0;  
                   let " <f> " : TrObj -> R = " h " in  
                   while(!<i> " <sum> " ;  
                   i := !<i> + 1;  
                   sum div " u " .size;}]}}}}}}

Composition : {λG.  
               {λF.  
                {let g = {G} in  
                 {let f = {F} in  
                  {box [{"let " " <g1> " : TrObj -> R = " g " in  
                          let " <f1> " : #gamma.varReal # -> " gamma.varReal # = " f " in  
                          fn " <z> " : TrObj => (" <f1> " (<g1> " <z> " ))]}}}}}}

TrObjToCoord : {λTr.  
                {λPos.  
                 {λCdn.  
                  {let tr = {Tr} in  
                   {let pos = {Pos} in  
                    {let cdn = {Cdn} in  
                     {box [{"fn " "TrObjToCoordinate" : TrObj => " cdn"("pos"("tr"("<x.TrObjToCoordinate>")))]}]]}}}}}}

maply : {λF.  
         {λz.  
          {let f = {F} in  
           {let u = {z} in  
            {box [{f u}]}]]}}

distance : {λP1.  
              {λP2.  
               {let p1 = {P1} in  
                {let p2 = {P2} in  
                 {box [{"let (x1, y1)=" p1 " in let (x2, y2)="p2" in  
                         Sqrt((x2-x1)^2+(y2-y1)^2)]}]]}}]}
c.1.3 Synthesized L1-Program

We executed the following inhabitation questions which resulted in the corresponding inhabitants (Many more similar inhabitation requests can be thought of.). The given programs resulted from reducing the inhabitants in L2. All programs were synthesized within a few milliseconds.

1. \([- \ ? : \#[R,\text{Distance}]] \) (asking for code of a distance value)
   - Inhabitant:
     \[
     \text{distance}(\text{box } ["\text{cdn(\text{pos(TrV(O))})}"], \text{box } ["\text{cdn(\text{pos(TrV(O))})}"])
     \]
   - Program:
     \[
     \text{box}\ \text{let}\ (x1, y1) = \text{cdn(\text{pos(TrV(O))})}\ \text{in}\ \text{let}\ (x2, y2) = \text{cdn(\text{pos(TrV(O))})}\ \text{in}
     \sqrt{(x2-x1)^2+(y2-y1)^2)}
     \]

2. \([- \ ? : \#([R,\text{Fh,ms}]) \) (asking for code of a value that is a measurement in Fh)
   - 11 Inhabitants were found. We only show one of them. They use the various conversion functions provided in the repository. Interestingly, also some average temperatures in Fahrenheit are found (using combinator A).
     \[
     \text{mapply}<\alpha.\text{varReal}->\text{R}>(\text{box } ["\text{builtInCel2Fh}"], \text{mapply}<\alpha.\text{varReal}->\text{R}>(\text{Bullet(\text{box } ["\text{tmp}"], \text{box } ["\text{TrV}"]), \text{box } ["O"])))
     \]
   - The corresponding program is:
     \[
     \text{box}\ \text{builtInCel2Fh}n\ x : \text{TrOb}j = > (\text{tmp(TrV x)})O
     \]

3. \([- \ ? : \#([R,\text{Cel,Avg,ms}]) \) (asking for code of an average value that is a measurement in Cel)
   - Inhabitants:
     \[
     \text{A(Bullet(\text{box } ["\text{tmp}"], \text{box } ["\text{TrV}"]), \text{box } ["V"])}
     \]
   - Program:
     \[
     \text{box}
     \text{let}\ x : \text{ref int} = \text{ref 0};
     \text{let}\ y : \text{ref R} = \text{ref 0};
     \text{let}\ z : \text{TrOb}j -> R = \text{fn v : TrOb}j = > (\text{tmp(TrV v)})\ \text{in}
     \text{while}(!V.size)\ \text{do}
     \text{sum} = : (x(V[!i]))+y;
     x := x+1;
     \text{sum div V.size};
     \]
C.2 SORTING ROUTINES

We discuss the synthesis of sorting routines.

C.2.1 Type Repository

The type repository is as follows:

**base:** int, bool, R, rational, node, Graph

**D1:** incTO, decTO, DAG, PO

**kinds:**
- varComp ~> R, int, rational, node
- varOrder ~> incTO, decTO, PO

**subtypes:**
- int <= R, int <= rational, rational <= R

**combinators C**
- lessThan : \([\text{pair}(R,R)\to\text{bool},\text{incTO}]\)
- edges2PartialOrder : \([\text{Graph},\text{DAG}]\to\text{pair}(\text{node},\text{node})\to\text{bool},\text{PO}]\)
- graph : \([\text{Graph},\text{DAG}]\)
- graph2node : \(\text{Graph}\to\text{Array}(\text{node})\)

**combinators D**
- S : \(#(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool})\to\text{ #(Array(\alpha.\text{varComp})\to\text{Array(\alpha.\text{varComp})})},\)\)
- #stop(\alpha.\text{varOrder}) \to \text{ #(\Omega\to\text{Sorted(\alpha.\text{varOrder})})}\)
- swap : \(#(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool})\to\text{ #(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool})},\)\)
- #a.\text{varOrder} \to \#\text{Rev}(a.\text{varOrder})\)
- Phi : \(#(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool})\to\text{ #(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool})},\)\)
- #\text{Rev}(\text{incTO}) \to \#\text{stop}(\text{decTO})\)
- #\text{Rev}(\text{decTO}) \to \#\text{stop}(\text{incTO})\)

**stopWrapper** : \(#(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool}),\)\)
- e.\text{varOrder} \to \#(\text{pair}(\alpha.\text{varComp},\alpha.\text{varComp})\to\text{bool})\),\)
- \text{stop}(e.\text{varOrder})\]

**mapply** : \(#(\text{Array(\text{node})})\to[\text{Array(\text{node})},\text{Sorted(PO)}])\to\text{ #(Array(\text{node})\to[\text{Array(\text{node}),\text{Sorted(PO)}]})\]

\[-? : #(\text{Array(R)})\to[\text{Array(R),\text{Sorted(PO)}]}\]

57
c.2.2 L2-Implementations

The following L2-Implementations are given to the combinators:

declarations
S : \{\lambda \text{Order}. \{\text{letbox order} = \{\text{Order}\} \text{ in } \{\text{box}\} \} \}
let orderFunction : ("#\alpha\text{.varComp#","#\alpha\text{.varComp#"}) -> bool = "order"
in
let swapInArray : (Array("#\alpha\text{.varComp#"),int,int) -> Array("#\alpha\text{.varComp#")) =
fn ("<\text{Swap>"","Index","Index") : (Array("#\alpha\text{.varComp#"),int,int) =>
let temp : ref "#\alpha\text{.varComp#" := ">\text{Swap>"","Index") := "<\text{Swap>"","Index") := !temp;
in

let bubbleSort : Array("#\alpha\text{.varComp#")) -> Array("#\alpha\text{.varComp#")) =
fn "<arrayS>" : Array("#\alpha\text{.varComp#")) =>
let swapped : ref boolean = ref true;
while (swapped) do
swapped := false;
let i : ref int = ref 0;
while (!i < "<arrayS>".size) do
if not orderFunction("<arrayS>"[!i-1],"<arrayS>"[!i]) then
swapInArray("<arrayS>"[!i-1],!i);
swapped := true;
else
skip;
i := !i+1;
in
fn "<yS>" : Array("#\alpha\text{.varComp#")) => bubbleSort ("<yS>")

swap : \{\lambda \text{Order}. \{\text{letbox order} = \{\text{Order}\} \text{ in } \{\text{box}\} \} \}
fn (x,y) : ("#\alpha\text{.varComp#"","#\alpha\text{.varComp#") => "order" (y,x)
"})

Phi : \{\lambda \text{Order}. \{\text{letbox order} = \{\text{Order}\} \text{ in } \{\text{box}\} \}

stopWrapper : \{\lambda \text{Order}. \{\text{letbox order} = \{\text{Order}\} \text{ in } \{\text{box}\} \}

mapply : \{\lambda \text{F}. \{\text{letbox f = \{\text{F}\} in
\{\text{letbox u = \{x\} in } \{\text{box}\} \}
let functionToApply : Array(node) -> Array(node) = "f" \} in

58
C.2.3 Synthesized L1-Program

We asked the following inhabitation question in order to synthesize a program that sorts an array of reals in decreasing order:

\[-?:\#(\text{Array}(\text{R})\rightarrow[\text{Array}(\text{R}),\text{Sorted}\text{(decTO})])\]

With an execution time of 165ms it resulted in the following inhabitant:

\[S<\alpha.\text{varComp}->\text{R}>(\text{Phi<}\alpha.\text{varComp}->\text{R}>(\text{swap<}\alpha.\text{varComp}->\text{R}>(\text{box ["lessThan"]}))\]

Which was reduced to the following sorting routine.

```haskell
let orderFunction : (R,R) -> bool = fn (x,y) : (R,R) => lessThan (y,x)
let swappedInArray : (Array(R),int,int) -> Array(R)
   fn (x,leftIndex,rightIndex) : (Array(R),int,int) =>
   let temp : ref R := !x[rightIndex];
   x[rightIndex] := !x[leftIndex];
   x[leftIndex] := !temp;

let bubbleSort : Array(R) -> Array(R) =
   fn y : Array(R) =>
      let swapped : ref boolean = ref true;
      while(swapped) do
         swapped := false;
         let i : ref int = ref 0;
         while (!i < y.size) do
            if not orderFunction(y[!i-1],y[!i]) then
               swappedInArray(y, !i-1, !i);
               swapped := true;
            else
               skip;
            !i := !i+1;
         in
      fn z : Array(R) => bubbleSort (z);

Changing the inhabitation question to

\[-?:\#\text{-stop}([\text{Array}(\text{node}),\text{Sorted}(\text{PO})])\]

we obtained an inhabitant representing code of an array of nodes of a given graph that is sorted topologically:

\[\text{map}\text{apply}(S<\alpha.\text{varComp}->\text{node}>(\text{stopWrapper<}\alpha.\text{varComp}->\text{node}>(\text{box ["edges2PartialOrder(graph)"]},
\text{box ["graph2node(graph)"]}))\]

This inhabitant results in the following program:
Box let functionToApply : Array(node) -> Array(node) =
let orderFunction : (node, node) -> bool = edges2PartialOrder(graph)

let swapInArray : (Array(node), int, int) -> Array(node) =
  fn (x, leftIndex, rightIndex) : (Array(node), int, int) =>
    let temp : ref node := x[rightIndex];
    x[rightIndex] := x[leftIndex];
    x[leftIndex] := !temp;
  in

let bubbleSort : Array(node) -> Array(node) =
  fn y : Array(node) =>
    let swapped : ref boolean = ref true;
    while(swapped) do
      swapped := false;
      let i : ref int = ref 0;
      while (!i < y.size) do
        if not orderFunction(y![i-1], y![i]) then
          swapInArray(y, !i-1, !i);
          swapped := true;
        else
          skip;
        i := !i+1;
      in
    fn z : Array(node) => bubbleSort (z);
  in

functionToApply(graph2node(graph))

C.3 COMBINED SCENARIO

This section is concerned with the combined scenario presented in Chapter 8.

C.3.1 Type Repository

The combinators are typed as follows:

base : bool, TrObj, R
D1 : Filter, incTO, decTO,
  Cart, Gpst, Cel, Fh, ms, Avg
D2 : Conv

kinds
  varComp ~> R
  varTemp ~> Cel, Fh
  varProperty ~> ms, Avg
  varTr ~> TrObj
varOrder ~> incTO, decTO

subtypes
incTO<=TO, decTO<=TO

combinatorsC
// Code-combinators for tracking:
O : TrObj
V : Array(TrObj)
TrV : TrObj, ~> D[D[R,R], Cart, [R, Gpst, [R, Cel]]
tmp : D[D[R,R], R, [R, a.varTemp, m]], ~> [R, a.varTemp, m]
builtInCel2Fh : [[R, Cel, ms]], ~> [R, Fh, ms], Conv
// Code-combinators for sorting
lessThan : [pair(R,R), ~> bool, incTO]
greaterThan : [pair(R,R), ~> bool, decTO]

combinatorsD
Composition : #(alpha.varTr, ~> [R, a.varTemp, ms]) ~> 
[[a.varTemp, gamma.varComp, ms], ~> 
[b, varTemp, gamma.varComp, ms], Conv] ~> 
#(alpha.varTr, ~> [R, b.varTemp, ms])
Bullet : #(D[P(R,R), R, [R, b.varTemp, c.varProperty]], ~> [R, g.varTemp, c.varProperty]) ~> 
(alpha.varTr, ~> D[P(R,R), R, [R, b.varTemp]], ~> 
#(alpha.varTr, ~> [R, g.varTemp, c.varProperty])
c2fh : [[R, Cel, a.varProperty], ~> [R, Fh, a.varProperty], Conv]

Diamond : #([alpha.varTemp, gamma.varComp, ms], ~> 
[[alpha.varTemp, gamma.varComp], ~> 
[beta.varTemp, gamma.varComp, ms], Conv] ~> 
#(alpha.varTemp, ~> gamma.varComp, ms])
Avg : #(alpha.varTr, ~> [R, Cel, ms]) ~> #(Array(alpha.varTr), ~> [R, Cel, Avg, ms])
F : #(alpha.varTr, ~> bool) ~> #([Array(alpha.varTr), ~> Array(alpha.varTr), Filter]
S : #([Array(alpha.varComp), ~> Array(alpha.varComp), Filter] ~> 
#([pair(alpha.varComp, alpha.varComp), ~> bool]), ~> #(Array(alpha.varComp), ~> Array(alpha.varComp)), 
#(stop(a.varOrder), ~> #(Omega, ~> Sorted(a.varOrder)))
stopWrapper : #([pair(alpha.varComp, alpha.varComp), ~> bool], e.varOrder) ~> 
#([pair(alpha.varComp, alpha.varComp), ~> bool], stop(e.varOrder)]
swap : #([pair(alpha.varComp, alpha.varComp), ~> bool]) ~> 
#([pair(alpha.varComp, alpha.varComp), ~> bool], a.varOrder, ~> #(Rev(a.varOrder)])
Phi : #([pair(alpha.varComp, alpha.varComp), ~> bool]) ~> 
#([pair(alpha.varComp, alpha.varComp), ~> bool], 
#(Rev(incTO)), ~> #(stop(docTO), #(Rev(docTO)), ~> #(stop(inctO)])
largerThanAV2 : #([Array(alpha.varTr), ~> [alpha.varComp, Cel, Avg, ms]), ~> 
#([pair(alpha.varComp, alpha.varComp), ~> bool], stop(docTO)]) ~>
filterAndSort : #(alpha.varTr -> [alpha.varComp,Cel,ms]) ->
  [pair(alpha.varComp,alpha.varComp)->bool,stop(decTO)] ->
  (Arra y(alpha.varTr)->[alpha.varComp,Cel,Avg,ms]) ->
  (Arra y(alpha.varTr)->[alpha.varComp,Cel,Avg,ms]) ->
  [pair(alpha.varComp,alpha.varComp)->bool,stop(decTO)] ->
  (Arra y(alpha.varTr) -> [alpha.varComp,Cel,ms]) ->
  (Arra y(alpha.varTr) -> book)

filterAndSort : (Arra y(alpha.varTr) -> [alpha.varComp,Cel,ms]) ->
  (Arra y(alpha.varTr) -> [alpha.varComp,Cel,ms]) ->
  [pair(alpha.varComp,alpha.varComp)->bool,stop(decTO)] ->
  (Arra y(alpha.varTr) -> [alpha.varComp,Cel,ms]) ->
  (Arra y(alpha.varTr) -> book)

The L2-implementations are as follows. Note that the interplay between largerThanAvg and filterAndSort that was discussed in Chapter 8 can now be retraced because the L2-implementation of the two combinators contain the higher-order computations mentioned.

declarations

Bullet : {lambda G, [lambda F, [letbox f = {F} in
  {letbox g = {G} in {box f”fn ” y_decl” : ” alpha.varTr” => ” g ” f ” y_decl ”)”
  ”]}}

cl2fh : {lambda z,[letbox u = {z} in {box f”let x : R = ” u ” in
  x * (9 div 5)+32 ”]}}

AverageFunction : {lambda TrzTemp, [letbox trzTemp = {TrzTemp} in {box f”let count : refInt = ref 0;
  let sum : ref R = ref 0;
  while(!count<”<TrzObjectArray>”.”size) do
    sum:="trzTemp”.”<TrzObjectArray>”.”[!count]”+sum;
    count:=!count+1;
    sum div ”<TrzObjectArray>”.”size;
in

"#:Arra y(alpha.varTr) ->
  #:alpha.varTr -> [alpha.varComp,Cel,ms] ->
  #:alpha.varTr -> book

filterAndSort : #:alpha.varTr -> [alpha.varComp,Cel,ms] ->
  #:pair(alpha.varComp,alpha.varComp)->bool,stop(decTO) ->
  [#(alpha.varTr -> [alpha.varComp,Cel,ms]) ->
    [#(alpha.varTr -> [alpha.varComp,Cel,ms]) ->
      #(alpha.varTr -> book)]

|- ? : #:Array(TrObj) -> [Array(TrObj),Sorted(decTO)]
fn "<trOb jArra yMain>" : Array("#alpha.varTr#") =>
calcAvgTemperature("<trOb jArrayMain>")
"]]"

F : [lambda Predicate.
    [letbox predicate = {Predicate} in {box !}]
    fn "<xF>" : Array("#alpha.varTr#") =>
        let newArra y : ref Array("#alpha.varTr#") = new Array("#alpha.varTr#");
        let ctrSource : ref in t = ref 0;
        let ctrTarget : ref in t= ref 0;
        while (!ctrSource."<xF>".size) do
            if "predicate"("<xF>"[!ctrSource]) then
                newArra y[!ctrTarget]="<xF>"[!ctrSource]
                ctrTarget:=!ctrTarget+1
            else
                skip
                ctrSource:=!ctrSource+1
        newArra y
    "]]"

swap : [lambda Order.[letbox order = {Order} in {box !}]
    fn (x,y) : ("#alpha.varComp#","#alpha.varComp#") => "order" (y,x)
    "]]"

Phi : [lambda Order.[letbox order = {Order} in {box [order]}]}

largerThanAv : [lambda AverageFunction.
    [lambda Order.
        [lambda Tr2TmpFunction.
            [letbox av = {lambda AvFn,[lambda Ar,[letbox avFn = {AvFn} in
                [letbox ar = {Ar} in {box[avFn("ar")])}]} AverageFunction Array] in
            [letbox tr2TmpFn = {Tr2TmpFunction} in {box !]
                fn "<x_largerThanA v2>" : ("#alpha.varTr#","#alpha.varTr#") =>
                    not("order"("av","tr2TmpFn"("<x_largerThanAv2>")))
            "]]]]}}]

filterAndSort :
    [lambda ConvertTrObjToTemp.
        [lambda Order.
            [lambda CalculateAverageTempGenerator.
                [lambda PredConstructor.
                    [lambda FilterGenerator.
                        [letbox trObj2Temp = {ConvertTrObjToTemp} in
                            [letbox calcAvgTemp = {CalculateAverageTempGenerator box ["tr2Tmp_filterAndSort"]} in
                                [letbox pred = {PredConstructor box ["calcAvgTempFunction_filterAndSort"]} in
                                    box["orderFunction"] box["<trOb jArray_filterAndSort>"]
                                    box["tr2Temp_filterAndSort"] in
                                [letbox filter = {FilterGenerator box ["averageFilterPredicate"]} in
                                    box["filterAndSort"] box["<trOb jArray_filterAndSort>"]
                                    box["tr2Temp_filterAndSort"] in
                "]]]]}]}})}}]

63
let box order = {Order} in {box ["fn "<trOb jArray_filterAndSort"> : Array(" #alpha.varTr# ") =>
  let tr2Temp_filterAndSort : "#alpha.varTr#" -> "#alpha.varComp#" = "trOb j2T emp"
  in
  let orderFunction : (" #alpha.varComp# "," #alpha.varComp# ") -> bool = "order"
  in
  let calcAvgTempFunction_filterAndSort :
    Array(" #alpha.varTr# ") -> "#alpha.varComp#" = "calcAvgTemp"
  in
  let averageFilterPredicate : "#alpha.varTr#" -> bool = "pred"
  in
  let filterByAverage_filterAndSort :
    Array(" #alpha.varTr# ") -> Array(" #alpha.varTr# ") = "filter"
  in
  let swap : (Array(" #alpha.varTr# "),int,int) -> Array(" #alpha.varTr# ") = fn ("<xSw ap>",leftIndex,rightIndex) : (Array(" #alpha.varTr# "),int,int) =>
    let temp : ref " #alpha.varTr# " := "<xSw ap>"[rightIndex];
    "<xSw ap>"[rightIndex] := "<xSw ap>"[leftIndex];
    "<xSw ap>"[leftIndex] := !temp;
    in
  let bubbleSort : Array(" #alpha.varTr# ") -> Array(" #alpha.varTr# ") = fn ("<arrayS>") : Array(" #alpha.varTr# ") =>
    let swapped : ref boolean = ref true;
    while(swapped) do
      swapped := false;
      let 1 : ref int = ref 0;
      while (!1 < "<arrayS>".size) do
      if not orderFunction(tr2Temp_filterAndSort("<arrayS>"[!1-1]),
        tr2Temp_filterAndSort("<arrayS>"[!1])) then
        swap("<arrayS>", !1, !1);
        swapped := true;
      else
        skip;
        i := !i+1;
      in
    let "<filteredTrOb jArray_filterAndSort>" : Array(" #alpha.varTr# ") =
      filterByAverage_filterAndSort("<trOb jArray_filterAndSort>")
    in
    bubbleSort("<filteredTrOb jArray_filterAndSort>")
  \]}}}}}}}}}}}}}}
c.3.3 Synthesized L1-Program

The inhabitant for the inhabitation question

\[- \ ; \ : \ #(\text{Array}(\text{TrObj})-\rightarrow[\text{Array}(\text{TrObj}), \text{Sorted}(\text{decTO})]) \]

was

\[
\text{filterAndSortFunction}<\alpha.\text{varComp}-\rightarrow\alpha.\text{varTr}-\rightarrow\text{TrObj}> (  
\text{Bullet}<\alpha.\text{varTr}-\rightarrow\text{TrObj}> (\text{box} \ ["tmp"], \text{box} \ ["TrV"]),  
\text{stopWrapper}<\alpha.\text{varComp}-\rightarrow\text{R} \rightarrow(\text{box} \ ["\text{greaterThan}"]),  
\text{AverageFunction}<\alpha.\text{varTr}-\rightarrow\text{TrObj}>,  
\text{largerThanAv2}<\alpha.\text{varComp}-\rightarrow\text{R},\alpha.\text{varTr}-\rightarrow\text{TrObj}>,  
\text{F}<\alpha.\text{varTr}-\rightarrow\text{TrObj}>)
\]

The synthesis took 9624 ms (00:00:09.624886) on the test system. 9 elements were added to the FailCache and 11 to the SuccessCache during the execution. The execution graph contains 42 nodes and 43 groups. The memory consumption was less than 12 MB. Figure C.1 depicts the execution graph in a Cluster View.
Figure C.1: Execution graph of the combined example
The inhabitant has the term structure depicted in Figure C.2.

The program resulting from this inhabitant is:

```ocaml
box
  fn x : Array(TrObj) =>
    let tr2Temp_filterAndSort : TrObj -> R => fn y : TrObj => (tmp(TrV y))
  in

let orderFunction : (R,R) -> bool =
  fn (x,y) : (R,R) => lessThan (y,x)
  in

let calcAvgTempFunction_filterAndSort : Array(TrObj) -> R =
  let calcAvgTemperature : Array(TrObj) -> R =
    fn z : Array(TrObj) =>
      let count : ref int = ref 0;
      let sum : ref R = ref 0;
      while(!count<z.size) do
        sum:=tr2Temp_filterAndSort(z[!count])=sum;
        count:=!count+1;
        sum div z.size;
      in
      fn v : Array(TrObj) => calcAvgTemperature(v)
  in

let averageFilterPredicate : TrObj -> bool =
  fn w : TrObj => not(orderFunction(calcAvgTempFunction_filterAndSort(x),
    tr2Temp_filterAndSort(w)))
  in

let filterByAverage_filterAndSort : Array(TrObj) -> Array(TrObj) =
  fn u : Array(TrObj) =>
    let newArray : ref Array(TrObj) = new Array(TrObj);
    let ctrSource : ref int = ref 0;
```
let ctrTarget : ref int = ref 0;
while (!ctrSource < u.size) do
    if averageFilterPredicate(u[ctrSource]) then
        newArray[ctrTarget] = u[ctrSource]
        ctrTarget := !ctrTarget + 1
    else
        skip
        ctrSource := !ctrSource + 1
in

let swap : (Array(TrObj),int,int) -> Array(TrObj) = fn (x',leftIndex,rightIndex) : (Array(TrObj),int,int) =>
    let temp : ref TrObj := x'[rightIndex];
    x'[rightIndex] := x'[leftIndex];
    x'[leftIndex] := !temp;
in

let bubbleSort : Array(TrObj) -> Array(TrObj) = fn y' : Array(TrObj) =>
    let swapped : ref boolean = ref true;
    while (swapped) do
        swapped := false;
        let i : ref int = ref 0;
        while (!i < y'.size) do
            if not orderFunction(tr2TemporalAndSort(y'![i-1]),
                                tr2TemporalAndSort(y'![i])) then
                swap(y', !i-1, !i);
                swapped := true;
            else
                skip;
            i := !i + 1;
in
    in

let z' : Array(TrObj) = filterByAverage_filterAndSort(x)
in
bubbleSort(z')
The experiments were performed on a single test system and not on a computing cluster to make measurements more controllable and reproducible. The test system has the following specifications:

- Microsoft Windows 8 Enterprise (Version 6.2.9200 Build 9200)
- Processor Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz, 2661 MHz, 4 Core(s)
- Installed physical memory (RAM) 8 GB
- Microsoft Visual Studio Ultimate 2012 – Version 11.0.60610.01 Update 3
- Microsoft .NET Framework – Version 4.5.50743
BIBLIOGRAPHY


