Towards Combinatory Logic Synthesis

Jakob Rehof

Faculty of Computer Science
Technical University of Dortmund
jakob.rehof@cs.tu-dortmund.de

Abstract

A theory of composition synthesis based on inhabitation in combinatory logic is described and illustrated with examples. Composition synthesis automatically generates applicative combinations of typed combinators inhabiting a goal type. In the intended applications, combinatory type environments model repositories of components, and types are used to specify their interfaces enriched with semantic and behavioural information. The inhabitation relation in combinatory logic is seen to define an abstract logic programming language, by regarding type environments as logic programs whose operational semantics are given by a search procedure for inhabitants. Intersection types are used to specify the intended semantics and behaviour of combinators. The idea of composition synthesis based on combinatory logic has been implemented in a system, **CLS** (Combinatory Logic Synthesizer).

1. Introduction

In recent papers together with B. Düdder, M. Martens and P. Urzyczyn [8, 18] we have investigated computational aspects of the theory of inhabitation in variants of combinatory logic [13]. From the standpont of applications, this work is motivated by the idea that inhabitation in combinatory logic could be used as a foundation for automatic composition synthesis, that is, synthesis of compositions from a repository of software components. The basic idea here is that we can regard a combinatory type judgement

$$\Gamma \vdash e : \tau$$

as modeling the fact that combinatory expression e can be obtained by composition from a repository, Γ, of components which are exposed as combinator symbols and whose interfaces are exposed as combinator types. The decision problem of inhabitation, often indicated as

$$\Gamma \vdash ? : \tau$$

is the question whether a combinatory term e exists such that $\Gamma \vdash e : \tau$. Such a term e is called an *inhabitant* in type τ. An algorithm (or semi-algorithm) for solving the inhabitation problem searches for inhabitants and could be used to generate inhabitants e.

Under the propositions-as-types correspondence, inhabitation is the question of provability in a Hilbert-style presentation of a propositional logic, where Γ represents a propositional theory, τ represents a proposition to be proved, and e is a proof.

Since the repository represented by Γ may change, we need to consider a generalized form of combinatory inhabitation, referred to as *relativized inhabitation*. Under relativized inhabitation the environment Γ is not held fixed, in contrast to standard combinatory logic which usually considers a fixed base (such as $\mathbf{S}, \mathbf{K}$). It turns out that the relativized inhabitation relation is extremely expressive. In fact, it can be used to define a Turing-complete notion of computation, even in simple types (propositions). Starting from this observation we explore the viewpoint that the inhabitation relation can be seen as an abstract logic programming semantics in which combinatory type environments Γ play the role of programs, goal types τ with intersection types are synthesis specifications, and inhabitants e are programs synthesized by combinatory composition and satisfying the goal specification. Intersection types are introduced to represent abstract semantic and behavioural information in specifications.

The paper is organized as follows. In the first part of the paper (Section 2) we review the theoretical background on relativized inhabitation in combinatory logic. In the middle part (Section 3) we explain the design of a synthesis methodology based on inhabitation in combinatory logic. An inhabitation algorithm based on [8, 18] is described. The algorithm has been implemented by B. Düdder and M. Martens, with extensions and optimizations, within a composition synthesis framework **CLS** (Combinatory Logic Synthesizer). The last part of the paper (Section 4) contains a catalogue of illustrative examples of composition synthesis.

We should emphasize that this paper focuses entirely on the logical foundations and the methodology of composition synthesis. In particular, no attempt is made to describe the optimizations and extensions in **CLS** since doing so would not be possible within the limits of the paper. Furthermore, the ideas described here are the basis of ongoing and future work to continuously optimize, extend and conduct experiments with **CLS** in composition synthesis.

2. Combinatory logic and inhabitation

2.1 Standard combinatory logic with simple types

Standard simple typed combinatory logic (CL) is based on a language of applicative terms e,

$$e ::= X \mid (e\ e)$$

where $X, Y, \ldots$ ranges over combinator symbols. Simple types $A, B, \ldots$ are defined by

$$A ::= \alpha \mid c \mid A \to B$$

where $\alpha, \beta, \ldots$ ranges over a set $\mathbb{V}$ of type variables and c ranges over a set $\mathbb{C}$ of type constants. In standard (typed or untyped) combinatory logic [13] one usually fixes a set (often called a *base*) Γ of combinators. In the typed case we represent Γ by a type environ-

BEAT '13 January 22 2013, Rome.

ment, a finite set of bindings of the form $(X : A)$. A well known base consists of just the two combinators $\mathbf{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$ and $\mathbf{K} : \alpha \to \beta \to \alpha$, equivalent to λ-calculus (both in the untyped case and the simple typed case). The type rules of CL are shown in Figure 1, where S denotes a type substitution. Combinator types are regarded as *type schemes* which are *implicitly polymorphic*, allowing instantiation under substitution of types for type variables – the rule (var) implements this. Under the propositions-as-types correspondence, the system corresponds to a Hilbert-style formulation of propositional intuitionistic (minimal) logic, characteristically based on just two principles of deduction, substitution of formulae into axiom schemes (var) and modus ponens ($\to$E).

For a fixed base Γ of typed combinators, we can consider the decision problem of *inhabitation*:

Given a type A, does there exist a combinatory term e with $\Gamma \vdash_s e : A$?

A term e with $\Gamma \vdash_s e : A$ is called an *inhabitant* in A. We shall often use the abbreviation

$$\Gamma \vdash ? : A$$

for the inhabitation predicate and the associated decision problem. In so far as the base Γ formalizes a propositional logic under the Curry-Howard isormorphism, the inhabitation problem is the *provability* problem for the logic: an inhabitant e is a proof of proposition A. Probably the most well known result on inhabitation in simple types is Statman's theorem [24], that provability in propositional intuitionistic logic is PSPACE-complete. By the Curry-Howard isomorphism, so is the inhabitation problem for simple typed λ-calculus. And, by the equivalence of CL and simple typed λ-calculus, so is the inhabitation problem

$$\left\{ \begin{array}{l} \mathbf{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma), \\ \mathbf{K} : \alpha \to \beta \to \alpha \end{array} \right\} \vdash_s ? : A$$

for CL.

2.2 Relativization and Linial-Post theorems

This section could also have been titled: *"Undecidability of simple types"*. If the previous sentence suggests to the reader that *simple types might not be simple*, then it would already have served a good part of its purpose!

The impression that simple types are simple is, of course, not altogether wrong. But it is only right from certain perspectives. The *typability* problem for simple types is, indeed, simpler than most (it is linear time decidable). The equivalence results on λ-calculus and CL mentioned in the previous section might lead to the impression that combinatory logic and λ-calculus are really the same subject and that *provability* (inhabitation) questions in simple types are therefore essentially captured by Statman's theorem. Finally, viewed through the lense of logic, the subject is a fragment of propositional logic, and propositional logic is perhaps sometimes regarded as a comparatively trivial subject of *logic*.

Regarding the relation to λ-calculus, it should be observed that the base $\mathbf{S}, \mathbf{K}$ only represents a very special case of a set Γ of combinators, albeit an exceedingly important one. The moment we consider other combinatory bases (propositional theories) Γ, a much more complicated landscape is revealed.

A natural generalization of the fixed-base inhabitation problem is to consider the problem *relativized* to a theory Γ given as part of the input:

Given Γ and A, does there exist a combinatory term e with $\Gamma \vdash_s e : A$?

The relativized problem is, of course, at least as hard as any fixed-base problem. The relativized problem can be understood through a line of work beginning in 1946, when Tarski posed the question of decidability for arbitrary propositional axiom systems (often referred to as partial propositional calculi, abbreviated PPC). L. Linial and E. Post answered the question in 1948 [14]. In PPC-problems it is assumed that Γ consist of axioms that are tautologies. Linial and Post proved that there exists a PPC with an unsolvable decision problem (Linial-Post theorem). The result can be obtained by reduction from the word problem for semi-Thue systems [4, 28]. Since then, many results have been obtained for various PPC, e.g., Gladstone [11] and Singletary [22] showed that every r.e. degree can be represented by a PPC. In 1974, Singletary [23] showed that the implicational fragment of PPC can represent every r.e. many-one degree. The logical framework considered there is identical to combinatory bases in simple types. It follows that *the relativized inhabitation problem for simple types is undecidable*.

Later, various systems related to specific propositional logics have been studied by considering corresponding combinatory bases, including bases built from the combinators [25] $\mathbf{B}$, $\mathbf{B}'$, $\mathbf{C}$, $\mathbf{I}$, $\mathbf{K}$, $\mathbf{W}$. In some cases, related propositional logics turned out to have undecidable provability problems (see, e.g., [26]). In other cases, notably the implicational theory of ticket entailment, the problem of decidability turned out to be extremely difficult. The problem for ticket entailment, raised in 1975, appears to have been solved only very recently [1, 16]. In some of these works connections between the combinatory theory and λ-calculus are exploited, but it is worth while pointing out that combinatory problems are not likely to be all reducible to λ-calculi. For example, λ-calculi typically contain some form of the deduction theorem (abstraction), whereas this certainly might not be the case for an arbitrary combinatory theory.

2.3 Simple types: a Turing-complete programming language

The Linial-Post theorem suggests that we might be able to view simple types as a Turing-complete logic programming language based on the inhabitation relation. Under this perspective,

- *the input theory Γ is the program*, and
- *the search for an inhabitant is its execution*

We now show that this can easily be achieved, if we use the full freedom of relativization, i.e., we do not restrict ourselves to tautologous theories Γ and type theory, using type constructors freely. Consider a two-counter automaton $\mathcal{A} = \langle Q, q_0, q_F, \delta \rangle$ with a finite set of control states Q (start state q_0, accepting state q_F), counters $\mathtt{c}_1$ and $\mathtt{c}_2$ and transition relation δ given by the instructions $(i = 1, 2)$

- $\mathtt{q} : \mathtt{c}_i := \mathtt{c}_i + 1; \mathtt{goto\ p}$
- $\mathtt{q} : \mathtt{c}_i := \mathtt{c}_i - 1; \mathtt{goto\ p}$
- $\mathtt{q} : \mathtt{if}\ (\mathtt{c}_i = 0)\ \mathtt{then\ goto\ p\ else\ goto\ r}$

A configuration of the automaton is given by (q, n, m) with $q \in Q$, and $n, m \in \mathbb{N}$ representing the contents of $\mathtt{c}_1$ and $\mathtt{c}_2$, respectively. Two-counter automata are known to be Turing-complete.

Such an automaton can be represented by a combinatory environment $\Gamma_{\mathcal{A}}$ as shown in Figure 2. We assume type constants $\mathtt{0}_i$ and type constructors $\mathtt{Num}_i(\_)$, $\mathtt{s}_i(\_)$ $(i = 1, 2)$ representing numbers, type constants $\mathtt{q}$ representing elements of Q, a constructor $\mathtt{S}(\_)$ representing control state, a constant $\mathtt{Acc}$ representing acceptance, and constructors $\mathtt{C}_i(\_)$ $(i = 1, 2)$ representing the counters. The combinators $\mathtt{Z}_i, \mathtt{N}_i, \mathtt{Ld}_i$ $(i = 1, 2)$, $\mathtt{S[q]}$ and $\mathtt{Fin}$ are used to construct the initial and final configurations. A configuration (q, n, m) is represented as a type of the form

$$\mathtt{S(q)} \to \mathtt{C}_1(\mathtt{s}_1^n(\mathtt{0}_1)) \to \mathtt{C}_2(\mathtt{s}_2^m(\mathtt{0}_2)) \to \mathtt{Acc}$$

The instructions acting on counter $\mathtt{c}_1$ are represented by the combinators $\mathtt{Add}_1[\mathtt{q}, \mathtt{p}]$, $\mathtt{Sub}_1[\mathtt{q}, \mathtt{p}]$, $\mathtt{Tst}_1^{\mathtt{Z}}[\mathtt{q}, \mathtt{p}]$ and $\mathtt{Tst}_1^{\mathtt{NZ}}[\mathtt{q}, \mathtt{r}]$ as shown.

$$\frac{}{\Gamma, X : A \vdash_s X : S(A)}(\text{var}) \qquad \frac{\Gamma \vdash_s e : A \to B \quad \Gamma \vdash_s e' : A}{\Gamma \vdash_s (e\ e') : B}(\to\text{E})$$

Figure 1: Standard simple typed combinatory logic CL

Similar representations for instructions acting on counter $\mathtt{c}_2$ are assumed.

For an automaton $\mathcal{A}$ and initial configuration (q_0, n, m), let

$$\Gamma_{\mathcal{A}}(q_0, n, m) = \Gamma_{\mathcal{A}} \cup \{\mathtt{Q}_0 : \mathtt{q}_0, \mathtt{X}_1 : \mathtt{s}_1^n(\mathtt{0}_1), \mathtt{X}_2 : \mathtt{s}_2^m(\mathtt{0}_2)\}$$

It is now straight-forward to prove

THEOREM 1. *$\mathcal{A}$ accepts from the configuration (q_0, n, m) if and only if there exists e with*

$$\Gamma_{\mathcal{A}}(q_0, n, m) \vdash_s e : \mathtt{Acc}$$

So the inhabitation relation

$$\Gamma_{\mathcal{A}}(q_0, n, m) \vdash_s ? : \mathtt{Acc}$$

simulates $\mathcal{A}$.

For example, suppose that $\mathcal{A}$ is given by

$\mathtt{q}_0$: $\mathtt{c}_1 := \mathtt{c}_1 - 1$; goto $\mathtt{q}_1$
$\mathtt{q}_1$: if $(\mathtt{c}_1 = 0)$ then goto $\mathtt{q}_F$ else goto $\mathtt{q}_0$

and consider the initial configuration $(q_0, 1, 0)$. Then we have the combinators representing instructions in $\Gamma_{\mathcal{A}}$:

$\mathtt{Sub}_1[\mathtt{q}_0, \mathtt{q}_1]$: • $\mathtt{q}_0$: $\mathtt{c}_1 := \mathtt{c}_1 - 1$; goto $\mathtt{q}_1$
$(\mathtt{S}(\mathtt{q}_1) \to \mathtt{C}_1(\alpha) \to \mathtt{C}_2(\beta) \to \mathtt{Acc}) \to$
$(\mathtt{S}(\mathtt{q}_0) \to \mathtt{C}_1(\mathtt{s}_1(\alpha)) \to \mathtt{C}_2(\beta) \to \mathtt{Acc})$

$\mathtt{Tst}_1^{\mathtt{Z}}[\mathtt{q}_1, \mathtt{q}_F]$: • $\mathtt{q}_1$: if $(\mathtt{c}_1 = 0)$ then goto $\mathtt{q}_F$ else goto $\mathtt{q}_0$
$(\mathtt{S}(\mathtt{q}_F) \to \mathtt{C}_1(\mathtt{0}_1) \to \mathtt{C}_2(\beta) \to \mathtt{Acc}) \to$
$(\mathtt{S}(\mathtt{q}_1) \to \mathtt{C}_1(\mathtt{0}_1) \to \mathtt{C}_2(\beta) \to \mathtt{Acc})$

$\mathtt{Tst}_1^{\mathtt{NZ}}[\mathtt{q}_1, \mathtt{q}_0]$: $(\mathtt{S}(\mathtt{q}_0) \to \mathtt{C}_1(\mathtt{s}_1(\alpha)) \to \mathtt{C}_2(\beta) \to \mathtt{Acc}) \to$
$(\mathtt{S}(\mathtt{q}_1) \to \mathtt{C}_1(\mathtt{s}_1(\alpha)) \to \mathtt{C}_2(\beta) \to \mathtt{Acc})$

Fix $\Gamma_{\mathcal{A}}(q_0, 1, 0)$ and write $e : A$ as a shorthand for

$$\Gamma_{\mathcal{A}}(q_0, 1, 0) \vdash_s e : A$$

Then we have

$\mathtt{Ld}_1\ \mathtt{X}_1\ (\mathtt{N}_1\ \mathtt{Z}_1)$: $\mathtt{C}_1(\mathtt{s}_1(\mathtt{0}_1))$
$\mathtt{Ld}_2\ \mathtt{X}_2\ \mathtt{Z}_2$: $\mathtt{C}_2(\mathtt{0}_2)$
$\mathtt{S}[\mathtt{q}_0]\ \mathtt{Q}_0$: $\mathtt{S}(\mathtt{q}_0)$

Hence, we should have

$$\mathtt{Sub}_1[\mathtt{q}_0, \mathtt{q}_1]\ \mathcal{X}\ (\mathtt{S}[\mathtt{q}_0]\ \mathtt{Q}_0)\ (\mathtt{Ld}_1\ \mathtt{X}_1\ (\mathtt{N}_1\ \mathtt{Z}_1))\ (\mathtt{Ld}_2\ \mathtt{X}_2\ \mathtt{Z}_2) : \mathtt{Acc}$$

provided that we can find $\mathcal{X}$ with

$$\mathcal{X} : \mathtt{S}(\mathtt{q}_1) \to \mathtt{C}_1(\mathtt{0}_1) \to \mathtt{C}_2(\mathtt{0}_2) \to \mathtt{Acc}$$

In turn, we have

$$\mathcal{X} = \mathtt{Tst}_1^{\mathtt{Z}}[\mathtt{q}_1, \mathtt{q}_F]\ \mathcal{Y} : \mathtt{S}(\mathtt{q}_1) \to \mathtt{C}_1(\mathtt{0}_1) \to \mathtt{C}_2(\mathtt{0}_2) \to \mathtt{Acc}$$

provided we can find $\mathcal{Y}$ with

$$\mathcal{Y} : \mathtt{S}(\mathtt{q}_F) \to \mathtt{C}_1(\mathtt{0}_1) \to \mathtt{C}_2(\mathtt{0}_2) \to \mathtt{Acc}$$

But we do in fact have

$$\mathtt{Fin} : \mathtt{S}(\mathtt{q}_F) \to \mathtt{C}_1(\mathtt{0}_1) \to \mathtt{C}_2(\mathtt{0}_2) \to \mathtt{Acc}$$

so we can take $\mathcal{Y} = \mathtt{Fin}$. Hence, we have in sum that

$\mathtt{Sub}_1[\mathtt{q}_0, \mathtt{q}_1]$
$(\mathtt{Tst}_1^{\mathtt{Z}}[\mathtt{q}_1, \mathtt{q}_F]\ \mathtt{Fin})$
$(\mathtt{S}[\mathtt{q}_0]\ \mathtt{Q}_0)$
$(\mathtt{Ld}_1\ \mathtt{X}_1\ (\mathtt{N}_1\ \mathtt{Z}_1))$
$(\mathtt{Ld}_2\ \mathtt{X}_2\ \mathtt{Z}_2) : \mathtt{Acc}$

The combinatory expression shown obviously represents the program defining the automaton, instruction for instruction. Notice that computation is simulated by reversed implications in the combinatory logic, generating new goals (implicational antecedents) from current goals (implicational consequents). This is comparable to goal-directed search in logic programming.

2.4 Intersection types

Based on the previous observations we may begin to see combinatory types (type schemes), even simple ones, as a powerful specification (indeed, logic programming) mechanism under the inhabitation relation. However, this does not mean that it is necessarily very *easy* to specify desired programs this way. Naturally, it depends on the context and purpose. Before going further along this train of thought, let us observe that a simple looking (again, a deceptive appearance!) extension of simple types leads to entirely new possibilities.

Intersection types of the form $\tau \cap \tau'$ were introduced in the λ-calculus [3, 17]. An example of the expressive power of intersection types is that they can readily represent any finite function, at the type level. If $f : D \to D$ is a finite function we can represent f by the intersection type τ_f:

$$\tau_f = \bigcap_{d \in D} d \to f(d)$$

Something like this cannot be done in simple types and suggests that intersection types offer a powerful mechanism for specifying functional *behaviour* very directly. Another simple example showing the power of intersection types as a mechanism for specification is the following *unique specification* property for intersection types. It shows that we can specify any given combinatory term uniquely using intersection:

PROPOSITION 2 (Unique specification [18, 19]). *For every combinatory expression e there exists an environment Γ_e and a type τ_e such that e is the unique term with $\Gamma_e \vdash e : \tau_e$.*

Generally, intersection types are known to capture deep semantic properties of λ-terms. The system types exactly the strongly normalizing terms [3, 17], hence typability is undecidable. In contrast, typability in simple types is linear time decidable, so there is an enormous gap between simple types and intersection types from the perspective of typability. The inhabitation problem for λ-calculus with intersection types is closely related to the λ-definability problem [20, 21] and is also undecidable [27] (compared to PSPACE-completeness for simple types).

Intersection types have been introduced into combinatory logic [5], with preservation of the correspondence to λ-calculus, i.e., between the intersection typed base $\mathbf{S}, \mathbf{K}$ to λ-calculus with intersection types. As a consequence, both typability and inhabitation are undecidable for combinatory logic with intersection types, for the fixed base $\mathbf{S}, \mathbf{K}$.

Type expressions, ranged over by τ, σ etc., are defined by

$$\tau \ ::= \ \boldsymbol{\omega} \mid \alpha \mid \tau \to \sigma \mid \tau \cap \sigma$$

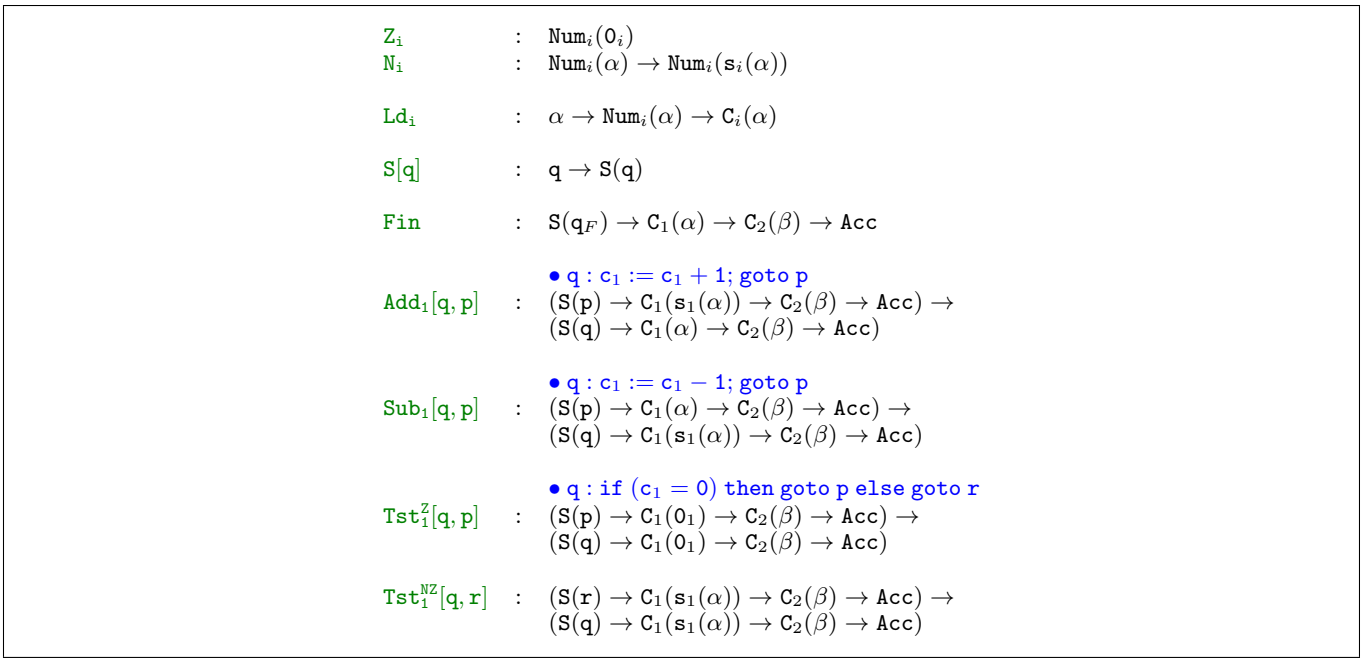


Figure 2: Construction of environment $\Gamma_{\mathcal{A}}$ for two-counter automaton $\mathcal{A}$

where $\boldsymbol{\omega}$ is a special type constant and α ranges over type variables as usual. Types are taken modulo commutativity ($\tau \cap \sigma = \sigma \cap \tau$), associativity ($(\tau \cap \sigma) \cap \rho = \tau \cap (\sigma \cap \rho)$), and idempotency ($\tau \cap \tau = \tau$). As a matter of notational convention, function types associate to the right, and $\cap$ binds stronger than $\to$. A type *environment* Γ is a finite set of type assumptions of the form $x : \tau$. A type $\tau \cap \sigma$ is said to have τ and σ as *components*. For an intersection of several components we sometimes write $\bigcap_{i=1}^{n} \tau_i$ or $\bigcap_{i \in I} \tau_i$ or $\bigcap\{\tau_i \mid i \in I\}$, where the empty intersection is identified with $\boldsymbol{\omega}$.

There is an interesting standard notion of subtyping for intersection types. Subtyping $\leq$ is the least preorder (reflexive and transitive relation) on types, with

- $\sigma \leq \boldsymbol{\omega}, \boldsymbol{\omega} \leq \boldsymbol{\omega} \to \boldsymbol{\omega}$
- $\sigma \cap \tau \leq \sigma, \sigma \cap \tau \leq \tau$
- $\sigma \leq \sigma \cap \sigma$
- $(\sigma \to \tau) \cap (\sigma \to \rho) \leq \sigma \to \tau \cap \rho$
- $\sigma \leq \sigma' \wedge \tau \leq \tau' \Rightarrow \sigma \cap \tau \leq \sigma' \cap \tau'$
- $\sigma \leq \sigma' \wedge \tau \leq \tau' \Rightarrow \sigma' \to \tau \leq \sigma \to \tau'$

We identify σ and τ when $\sigma \leq \tau$ and $\tau \leq \sigma$. The following distributivity properties follow from the axioms of subtyping:

$$(\sigma \to \tau) \cap (\sigma \to \rho) = \sigma \to (\tau \cap \rho)$$
$$(\sigma \to \tau) \cap (\sigma' \to \tau') \leq (\sigma \cap \sigma') \to (\tau \cap \tau')$$

Combinatory logic with intersection types can now be defined as shown in Figure 3.

2.5 Finite and bounded combinatory logic

From the standpoint of relativized inhabitation for combinatory logic, it would appear that there is no difference between simple types and intersection types: the inhabitation problems are equally undecidable. The enormous distance between the systems in terms of typability and provability for the fixed case of $\mathbf{S}, \mathbf{K}$ (linear time and PSPACE decidability vs. undecidability) suggests that relativized inhabitation provides a too coarse-grained lense to compare the systems.

In order to investigate the finer structure of the expressive power of combinatory logic and intersection types in the setting of relativized inhabitation, and in order to understand the algorithmics of decidable subsystems, we have considered restrictions of the relativized problem.

First, we have considered the restriction of relativized inhabitation, where combinator types are considered monomorphic [18]. Logically, this amounts to eliminating the schematic interpretation of axioms in a Hilbert-style system. The resulting system is called finite combinatory logic, FCL. Second, we have considered the generalization of FCL to arbitrary bounded levels of polymorphism, where a bound k is imposed on the depth of types allowed to instantiate combinator type schemes [8]. The resulting systems are referred to as bounded combinatory logic, BCL_k.

DEFINITION 3. *(Levels) Given an intersection type τ we define the* level *of τ, written $\ell(\tau)$, as follows.*

$$\begin{array}{lcl} \ell(a) & = & 0, \textit{ for } a \in \mathbb{C} \cup \mathbb{V}; \\ \ell(\tau \to \sigma) & = & 1 + \max\{\ell(\tau), \ell(\sigma)\}; \\ \ell(\bigcap_{i=1}^{n} \tau_i) & = & \max\{\ell(\tau_i) \mid i = 1, \ldots, n\}. \end{array}$$

For a type substitution S defined on a finite set $Dom(S)$ of type variables we define its level, denoted $\ell(S)$, by $\ell(S) = \max\{\ell(S(\alpha)) \mid \alpha \in Dom(S)\}$.

We can now define, for each $k \geq 0$, the systems of bounded combinatory logic with intersection types, denoted $\text{BCL}_k(\to, \cap)$, by the rules shown in Figure 4. In rule (var), the substitution S used to instantiate type schemes is restricted to be at most of level k. Finite combinatory logic with intersection types, $\text{FCL}(\to, \cap)$, arises by simplifying the (var) rule to $\Gamma, X : \tau \vdash x : \tau$. Both systems can also be restricted to simple types, yielding $\text{FCL}(\to)$ and $\text{BCL}_k(\to)$, respectively, by restricting types to be simple types, eliminating rule ($\cap$E) and restricting the subtyping relation appropriately.

The restrictions finitize the logic strictly and lead to decidable inhabitation problems for all systems. We can then study the complexity of inhabitation in these systems as a function of the restictions (monomorphic or bounds k) and derive results about relative expressive power.

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

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash_k e : \tau_2}{\Gamma \vdash e : \tau_1 \cap \tau_2}(\cap\text{I}) \qquad \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}(\leq)$$

Figure 3: Combinatory logic with intersection types $\text{CL}(\to, \cap)$

$$\frac{[\ell(S) \leq k]}{\Gamma, X : \tau \vdash_k X : S(\tau)}(\text{var}) \qquad \frac{\Gamma \vdash_k e : \tau \to \tau' \quad \Gamma \vdash_k e' : \tau}{\Gamma \vdash_k (e\ e') : \tau'}(\to\text{E})$$

$$\frac{\Gamma \vdash_k e : \tau_1 \quad \Gamma \vdash_k e : \tau_2}{\Gamma \vdash_k e : \tau_1 \cap \tau_2}(\cap\text{I}) \qquad \frac{\Gamma \vdash_k e : \tau \quad \tau \leq \tau'}{\Gamma \vdash_k e : \tau'}(\leq)$$

Figure 4: Bounded combinatory logic BCL_k

THEOREM 4 ([18],[19]). *For finite combinatory logic we have:*

1. *Relativized inhabitation in* $\text{FCL}(\to)$ *is in* PTIME
2. *Relativized inhabitation in* $\text{FCL}(\to, \cap)$ *is* EXPTIME*-complete*

THEOREM 5 ([8]). *For bounded combinatory logic we have:*

1. *Relativized inhabitation in* $\text{BCL}_k(\to)$ *is* EXPTIME*-complete for all* k
2. *Relativized inhabitation in* $\text{BCL}_k(\to, \cap)$ *is* $(k+2)$-EXPTIME*-complete*

Both theorems reveal a difference between the simple typed problem and the problem with intersection types. It is useful to understand the lower- and upper-bound techniques leading to these results, since they lead to an understanding of the possibilities and limits of algorithm design, but also because they lead to a better understanding of the expressive power of the inhabitation relation for combinatory logic with intersection types. In particular, they show how inhabitation is related to tree automata and space-bounded alternating Turing machines [2]. Space limitations preclude a very detailed survey, so we must refer the reader to the sources [8, 18, 19] for a fuller treatment.

For $\text{FCL}(\to, \cap)$, we obtain the EXPTIME lower bound by reduction from the intersection non-emptiness problem for tree automata. Essentially, for a set of tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_N$ and a function symbol f, every tree automaton rule R of the form $f(q_1, \ldots, q_m) \Rightarrow q$ is represented as a type $\tau_R = q_1 \to \cdots \to q_m \to q$, and all rules for the symbol f are collected using an intersection in the combinator type $f : \bigcap_R \tau_R$. One shows that non-emptiness of the intersection of languages recognized by the automata is equivalent to the inhabitation question $\Gamma \vdash ? : \bigcap_{i=1}^{N} q_{F_i}$, where q_{F_i} is the accepting state of $\mathcal{A}_i$.

The upper bound is achieved, in the case without subtyping, by reduction to non-emptiness of polynomial-sized alternating top-down tree automata. In the case with subtyping the upper bound is achieved using polynomial-space bounded alternating Turing-machines (ATMs, [2]).

For the bounded systems BCL_k, we prove the $(k+2)$-EXPTIME lower bounds by a generic simulation of alternating Turing machines operating in $\exp_{k+1}(n)$-bounded space, where $\exp_m$ denotes the iterated exponential function. For each k, we devise a numeral representation with intersection types in BCL_k for numbers between 0 and $\exp_{k+1}(n) - 1$, and we use this system to achieve a succinct representation (exploiting k-bounded polymorphism) of the Turing tape. By the $(k+2)$-EXPTIME lower bounds, the inhabitation problem where the bound k is taken as an input parameter is nonelementary recursive.

The upper bounds are achieved using alternating Turing machines [2] operating in $\exp_{k+1}(n)$-bounded space. Figure 5 shows an $\exp_{k+1}(n)$-space bounded algorithm for inhabitation in $\text{BCL}_k(\to, \cap)$ specified for an ATM. In this algorithm, simplicity has been prioritized over efficiency, so no attempt at optimization has been made beyond matching the $(k+2)$-EXPTIME lower bound. Since we have

$$\text{CL}(\to, \cap) = \bigcup_k \text{BCL}_k(\to, \cap)$$

the algorithm contains a semi-decision procedure for inhabitation in the unbounded system. The algorithm can be seen as an operational semantics for logic programming via the inhabitation relation, in the sense of Section 2.3 and Section 3.1 below.

For reasons of space, we must refer the reader to [8] for a detailed explanation of the ATM shown in Figure 5. Briefly and informally, the algorithm nondeterministically chooses a combinator $(X : \sigma)$ from Γ (line 1), then (line 2) constructs the intersection σ' of all instances of σ under substitutions whose ranges are limited to using variables and constants in Γ and τ and are bounded by k (this is the set $\mathcal{S}_x^{(\Gamma,\tau,k)}$). In line 3, a number m is chosen which shall be used as the arity of function types (paths) that can be exposed (under the algebra of $\leq$) in σ'. In line 4, a subset P of *paths* of the form $\pi = \tau_1 \to \cdots \to \tau_m \to \tau'$ (τ_i are called arguments and τ' is called the target of the path) are chosen from σ' (an intersection type can be written as an intersection of such paths). It is checked (line 5) that the intersection of targets τ' of the chosen paths is a subtype of the current goal type τ. If the paths have no argument types ($m = 0$) the algorithm accepts. Otherwise, all of the argument types for $i = 1 \ldots m$ need to be inhabited. Checking inhabitation of these types is done by a universal transition (line 8) where the goal type is reset, in parallel, to each of the argument types (line 9), and the algorithm loops back to the top (line 10).

3. Composition synthesis

In this section we explain the idea of using relativized inhabitation in combinatory logic with intersection types as a logical foundation for automatic composition synthesis.

3.1 The programme

Imagine we have a set of software components named $\texttt{F}_1, \ldots, \texttt{F}_\texttt{n}$. Each component has an interface type which is associated with the

$Input:\ \ \Gamma, \tau, k$
$loop:$
1 CHOOSE $(X : \sigma) \in \Gamma$;
2 $\sigma' := \bigcap\{S(\sigma) \mid S \in \mathcal{S}_x^{(\Gamma,\tau,k)}\}$;
3 CHOOSE $m \in \{0, \ldots, \|\sigma'\|\}$;
4 CHOOSE $P \subseteq \mathbb{P}_m(\sigma')$;

5 IF $(\bigcap_{\pi \in P} tgt_m(\pi) \leq \tau)$ THEN
6 IF $(m = 0)$ THEN ACCEPT;
7 ELSE
8 FORALL$(i = 1 \ldots m)$
9 $\tau := \bigcap_{\pi \in P} arg_i(\pi)$;
10 GOTO $loop$;
11 ELSE REJECT;

Figure 5: Alternating Turing machine deciding inhabitation in BCL_k

name of the component, $(\mathtt{F_i} : \tau_i)$. We can organize the collection of typed components $(\mathtt{F_1} : \tau_i)$ in a type environment Γ,

$$\Gamma = \{\mathtt{F_1} : \tau_1, \ldots, \mathtt{F_n} : \tau_n\}$$

We refer to such a collection of typed component names as a *repository*.

We can think of each named component interface $(\mathtt{F_i} : \tau_i)$ in a repository as a typed combinator symbol, and we can therefore ask the inhabitation question $\Gamma \vdash ? : \tau$. A combinatory inhabitant e is a program composed by applicative combination of the components in Γ. An inhabitation algorithm could therefore be used to *compute* such programs, and doing so would amount to automatic *synthesis* of the program e from Γ. We refer to this process as *composition synthesis*. The framework of relativized inhabitation in combinatory logic is the right logical model of this process, because the repository Γ may change.

In previous sections we have seen how types can be used to "program" the composition of combinators. In the remainder of this paper we shall try to illustrate how this idea can be used in composition synthesis from a repository by considering the collection of typed combinators as a kind of logic program which, when executed under the semantics of the inhabitation relation, synthesizes all possible compositions from the repository. We can detail the standpoint as follows:

- *the input repository Γ is a logic program (or "interface program") at the level of types*
- *the interface types of each component combinator in Γ are rules in this program*
- *the inhabitation goal τ is the input goal to the program*
- *the search for inhabitants is the execution of the program*
- *the inhabitants are programs generated by synthesis from the interface program as its solution space*

The correspondence between type theory and programming can be understood from the propositions-as-types correspondence. An inhabitant e in $\Gamma \vdash e : \tau$ is a logical proof of the proposition τ from the theory Γ, and it is a program of type τ. The notion of logic programming used here is broadly related to the idea of abstract logic programming languages of Miller et. al. [15], a proof theoretic generalization of logic programming where the declarative meaning of a logic program is defined by proof search.

3.2 Semantic specification

In applications we must assume that the components $\mathtt{F_i}$ are implemented in a usual programming language, for instance a Java-like language. We shall refer to the implementation language as the *native language*. For the sake of simplicity, in our examples we shall think of the native language as an applied λ-calculus (including recursion, standard functions, constants and type constructors, etc.) with simple types, but the basic methodology could essentially (perhaps with suitable extensions and modifications) be applied to most other relevant choices of native languages, provided there is a notion of function application.

The implemented components $\mathtt{F_i}$ have *native types* A_i as interfaces. These are just the types of the implemented components in the native language, in our examples simple types. But the native interfaces alone are typically not sufficient to "program" the interface for composition, because they *underspecify the semantics of the components* and thereby leave the solution space for composition too large. If all we know about a component $\mathtt{F}$ is that it has native type $\mathtt{int} \to \mathtt{int}$, we know very little about what $\mathtt{F}$ accomplishes. This is where intersection types come into play, as a mechanism for semantic specification.

As we have already indicated (Section 2.4 and Section 2.5), intersection types have enormous expressive power and can be used to describe the *behaviour* of programs, at different levels of abstraction. In applications to composition synthesis, we think of intersection types as hosting, in principle, a *two-level type system*, consisting of native types and semantic types. In distinction to native types, *semantic types* are abstract, application-dependent conceptual structures, drawn e.g. from a taxonomy (domain ontology). For example, we might consider a specification

$$\mathtt{F} : ((\mathtt{real}, \mathtt{real}) \cap \mathit{Cart} \to (\mathtt{real}, \mathtt{real}) \cap \mathit{Pol}) \cap \mathit{Iso}$$

where native types – above, the type $\mathtt{real}$ and the product type $(\mathtt{real}, \mathtt{real})$ – are extended by *superimposing* semantic types using intersection (in the example, Cart, Pol, Iso) expressing (relative to a given conceptual taxonomy) interesting domain-specific properties of the component combinator $\mathtt{F}$ — e.g., that it is an isomorphism transforming Cartesian to polar coordinates. More generally, we can think of semantic types as organized in any system of finite-dimensional feature spaces (e.g., Cart, Pol are features of coordinates, Iso is a feature of functions) whose elements can be mapped onto the native API using intersections, at any level of the type structure. The unique specification property (Proposition 2) shows that we can, in principle, specify the context of use of a combinator $\mathtt{F}$ as precisely as we want, given enough information in Γ.

We shall refer to a repository Γ of combinatory interfaces $(\mathtt{F} : \tau)$ with semantic type information τ as a *semantic repository*.

It is a possible advantage of the type-based approach to synthesis proposed here that types can be naturally associated with code,

because APIs already have (native) types onto which specifications can be superimposed. There is a natural semantics of composition which is logically founded in combinatory proof theory via the inhabitation relation. And specifications can be scaled to different levels of abstraction relative to given conceptual structures capturing domain knowledge.

For the method outlined above to make sense, we must require *native type correctness*, i.e., that every composition (inhabitant) generated be well-typed in the native language, since generated programs must be executed in the native language. We therefore require that for every combinatory interface $(\mathtt{F} : \tau)$ there exists a unique *underlying* native type A such that $A = \tau^\circ$, where $\_^\circ$ is an operation that removes all semantic type information, and such that A is a valid type of the implementation of $\mathtt{F}$ in the native language. Moreover, since we do not consider it practicable to verify semantic correctness of specifications against component implementations (in fact, we want only to rely on interface information in synthesis, independently of any further knowledge about the implementation), we *assume* but do not attempt to prove *semantic correctness*, i.e., that semantic types correctly describe the semantics of component implementations. In this strategy we follow [12], with similar motivations.

For example, suppose we have a component interface combinator $\mathtt{F} : [\mathtt{int}] \to [\mathtt{int}]$ (where $[\_]$ is the list type constructor in the native language). Judging from this native type alone, $\mathtt{F}$ could be doing just about anything to the input list. Suppose it is in fact a sorting function, pehaps implementing Quicksort. Then we should have a certain implementation program behind the component combinator $\mathtt{F}$, for example

$$
\begin{array}{l}
\mathtt{F} \triangleq \\
\mathtt{letrec\ Q} : [\mathtt{int}] \to [\mathtt{int}] = \\
\quad \lambda \mathtt{x} : [\mathtt{int}].\ \mathtt{if}\ (\mathtt{x} = \mathtt{nil})\ \mathtt{then\ nil} \\
\quad \mathtt{else} \\
\qquad (\mathtt{Q}\ (\mathtt{Filter}\ (\mathtt{tail\ x})\ (\lambda \mathtt{y}\text{: int}.(\mathtt{head\ x}) \leq \mathtt{y}))) :: \\
\qquad (\mathtt{head\ x}) :: \\
\qquad (\mathtt{Q}\ (\mathtt{Filter}\ (\mathtt{tail\ x})\ (\lambda \mathtt{y}\text{: int}.(\mathtt{head\ x}) > \mathtt{y})))
\end{array}
$$

In our semantic repository we only expose the typed component name in the form of a typed combinator symbol $(\mathtt{F} : \tau)$. We could, for example, have

$$\tau \quad = \quad ([\mathtt{int}] \to ([\mathtt{int}] \cap \mathit{Sorted})) \cap \mathit{SortingFunction.Quicksort}$$

where the notation $\sigma.\tau$ denotes the type τ together with the condition $\tau \leq \sigma$ (τ is a subtype of σ), so we can read $\sigma.\tau$ as "τ where $\tau \leq \sigma$".

By native type correctness we require that $\tau^\circ = [\mathtt{int}] \to [\mathtt{int}]$ and that the type $[\mathtt{int}] \to [\mathtt{int}]$ is in fact a valid native type of the implementation. And we assume, but do not attempt to prove, that the intended semantic interpretation of the type is true of the implementation (in this case, that $\mathtt{F}$ implements a sorting routine using the Quicksort algorithm which returns a sorted list).

3.3 Stratified system

In order to implement the strategy described in Section 3.2 it is useful to introduce a stratification of the type language reflecting the distinction between native and semantic types. In the stratified type grammar shown below, $\mathbb{T}_N$ comprises of native types which are simple types, $\mathbb{T}_S$ comprises of intersection types expressing semantic information, and $\mathbb{T}_U$ is the universe of types where semantic information can be superimposed onto native types with the intersection operator.

$$
\begin{array}{lcll}
\mathbb{T}_N & \ni & A, B & ::= \quad \alpha \mid c \mid A \to B \\
 & & & \qquad \alpha, \beta \in \mathbb{V}_N, c \in \mathbb{C}_N \\
\mathbb{T}_S & \ni & \varphi, \psi & ::= \quad \varepsilon \mid d \mid \boldsymbol{\omega} \mid \varphi \to \psi \mid \varphi \cap \psi \\
 & & & \qquad \varepsilon, \eta \in \mathbb{V}_S, d \in \mathbb{C}_S \\
\mathbb{T}_U & \ni & \tau, \sigma & ::= \quad A \mid \tau \to \sigma \mid \tau \cap \varphi
\end{array}
$$

We define the operation $\_^\circ$ on $\mathbb{T}_U$ by

$$
\begin{array}{lcl}
A^\circ & = & A \\
(\tau \to \sigma)^\circ & = & \tau^\circ \to \sigma^\circ \\
(\tau \cap \varphi)^\circ & = & \tau^\circ
\end{array}
$$

We lift the operation to type environments by

$$\Gamma^\circ = \{(\mathtt{F} : \tau^\circ) \mid (\mathtt{F} : \tau) \in \Gamma\}$$

Let the relation $\Gamma \Vdash e : \tau$ be defined by

$$\Gamma \Vdash e : \tau \quad \text{if and only if} \quad \Gamma \vdash e : \tau \quad \text{and} \quad \Gamma^\circ \vdash_s e : \tau^\circ$$

The relation $\Vdash$ captures the simultaneous inhabitation by e in the semantic and in the native system, thereby implementing the requirement of native type correctness.

The k-bounded relation $\Vdash_k$ is defined by using $\vdash_k$ (k-bounded versions of $\vdash$ and $\vdash_s$) in the definition. It is not difficult to modify the inhabitation algorithm shown in Figure 5 to implement the relations $\Vdash$ and $\Vdash_k$. Let us point out that $\Vdash_k$ and $\vdash_k$ have the same theoretical expressive power and complexity of inhabitation, and this would be true even if the native system is a decidable, simple typed finite combinatory logic (the reason can be found in [8], where the lower bound construction is regimented by what amounts to monomorphic simple native types built from a single base type denoted $\bullet$).

3.4 CLS

Inhabitation algorithms for the relativized relations

$$\Gamma \vdash_k ? : \tau \text{ and } \Gamma \Vdash_k ? : \tau$$

have been implemented at the technical university of Dortmund, by B. Düdder and M. Martens [6, 7, 9] in a system called **CLS** (Combinatory Logic Synthesizer) which is under continuous development. The implementation is integrated into a larger framework in which semantic repositories can be designed and exposed to the synthesis algorithm. The algorithm has also been deployed on a cloud computing platform with about 1200 compute cores. Experiments in combinatory logic synthesis based on the logical framework described in this paper are being conducted continuously in Dortmund, including composition synthesis of control programs for LEGO NXT robots and GUI-synthesis. The implementation of the inhabitation algorithm in **CLS** already contains a number of important optimizations and heuristics. Since it would transgress the limits of this paper to go into further details concerning the optimized algorithm and the environment of **CLS**, we refer the reader to [6, 7, 9] for more information and to forthcoming publications.

4. Example repositories

We illustrate the method for composition synthesis described above by examples. The examples can be solved with the **CLS** algorithm. We have attempted to illustrate different kinds of themes (taxonomy, parameterization, service composition, protocol, logic) of immediate interest in specification. We stick to relatively simple scenarios that can be stated briefly, and the examples do not in any way explore the full theoretical expressive power of the specification language.

4.1 Taxonomy

We illustrate the use of taxonomic conceptual structure represented in semantic types with subtyping. This example is taken from [9]. Consider the section of a repository of functions shown in Figure 6, where the native API of a tracking service is shown, where the native type R abbreviates the type real.

The intended meaning and use of the repository is as follows. The function Tr can be called with no arguments and returns a datastructure of type D((R, R), R, R) which indicates the position of the caller at the time of call and the temperature at that position. Thus, the function Tr could be used by a moving object to track itself and its temperature as it moves. The tracking service might be useful in an intelligent logistics application, where an object (say, a container) keeps track of its own position (coordinates at a given point in time) and condition (temperature). Thus, the first component of the structure D (a pair of real numbers) gives the 2-dimensional cartesian coordinate of the caller at the time of call, the second component (a real number) indicates the time of call, and the third component (a real number) indicates the temperature.

In addition to the tracking function Tr the repository contains a number of auxiliary functions which can be used to project different pieces of information from the datastructure D, with pos returning the position (coordinate and time), fst and snd project components of a coordinate, and tmp projects the temperature. Finally, there are conversion functions, cc2pl and cl2fh, which convert from cartesian to polar coordinates and from Celsius to Fahrenheit, respectively.

The problem with the standard, native API shown in Figure 6 is that it does not express any of the semantics of its intended use as described above. The example is radicalized by using only one native base type R for illustrative purposes. If we ask for an inhabitant of the type R, we will get an excessive number of solutions, most of which will make no semantic sense, as the reader can easily verify. We can solve this problem by introducing conceptual structure to capture the intended semantics. Figure 7 shows one such possible structure, which is intended to capture the semantics explained informally for our example above. The structure is given in the form of a taxonomic tree, the nodes of which are *semantic type names*, and where dotted lines indicate structure containment (for example, elements of the semantic type *TrackData* contain elements of semantic type *Pos* and *Temp*), and solid lines indicate subtyping relationships (for example, *Cart* and *Polar* are subtypes of *Coord*). We are assuming a situation in which certain semantic types can be represented in different ways (as is commonly the case), e.g., we have *Time* either as GPS Time (*Gpst*) or as Universal Time (*Utc*), we have temperature (*Temp*) either in Celsius (*Cel*) or in Fahrenheit (*Fh*), and coordinates can be either polar or cartesian.

In Figure 8 we show the repository of Figure 6 with semantic types superimposed onto the native API using intersection types. The superposition of semantic information can be considered as an annotation on the native API. As can be seen, the tracking combinator Tr uses a representation in which coodinates are cartesian, time is GPS, and temperature is Celsius. Level-0 polymorphic type variables (ε, η) are used to succinctly capture semantic information flow, e.g., the combinator pos projects a position (*Pos*) from a D-typed argument while preserving the semantic information attached to the component types (ε standing for the semantic qualification of the coordinate component, η for that of the time component). The types should be readily understandable given the previous explanation of the intended meaning of the API. Notice how we use intersection types to refine [10] semantic types, as for instance in the type of fst, where the type $(\mathit{Cart} \to \mathit{Cx}) \cap (\mathit{Polar} \to \mathit{Radius})$ refines the action of fst on the semantic type *Coord*.

With the semantically enriched API shown in Figure 8 considered as a combinatory type environment Γ_1 we can now ask meaningful questions that can be formalized as synthesis (inhabitation) goals. For example, we can ask whether it is possible to synthesize a computation of the current radius (i.e., the radial distance from a standard pole at the current position) by considering the inhabitation question $\Gamma_1 \Vdash ? : \mathtt{R} \cap \mathit{Radius}$. Sending this question to our inhabitation algorithm gives back the (in this case unique) solution

$$\Gamma_1 \Vdash \mathtt{fst}\ (\mathtt{cc2pl}\ (\mathtt{cdn}\ (\mathtt{pos}\ (\mathtt{Tr}())))) : \mathtt{R} \cap \mathit{Radius}$$

The example shows the automatic insertion of data type transformations (cc2pl) enabled by semantic type structure.

Naturally, the expressive power and flexibility of a repository depends on how it is designed and its type structure axiomatized ("programmed", referring to the logic programming analogy mentioned above), and we do not anticipate that our methodology will be applicable to repositories that have not been designed accordingly.

4.2 Parameterization

The following example illustrates parameterization in the form of higher-order functions and polymorphism, and the use of intersection types to capture functional relations on abstract types. In the semantic repository Γ_2 shown in Figure 9 we have the interface of a generic sorting function S, parameterized by a filter function (which filters the elements of a list by a predicate of type $\alpha \to \mathtt{bool}$ given as argument) and an order relation on α of type $\alpha \to \alpha \to \mathtt{bool}$. The specification for S says that, if given a total order on α, the result will be a sorted list, whereas a topologically sorted list will result if the given order is partial. It is easy to implement such a function S, or course. In addition, there is a component G which transforms a directed graph of α-typed nodes into the partial order on nodes induced by the edge relation of the graph. The combinator N gives a list of the nodes of the argument graph. Finally, there are two general composition combinators, $\circ$ and $\diamond$ (we shall use obvious infix notation for those in the example). The reader may recognize $\diamond$ and $\circ$ by their types as the standard combinators $\mathbf{S} = \lambda x.\lambda y.\lambda z.xz(yz)$, and $\mathbf{B}' = \lambda x.\lambda y.\lambda z.y(xz)$, respectively. We assume the semantic structure shown in Figure 10.

If we ask if we can synthesize a program computing a topologically sorted list of nodes from a given input graph, we can consider the inhabitation goal

$$\Gamma_2 \Vdash ? : \mathtt{Graph}(\alpha) \to ([\alpha] \cap \mathit{TopSorted})$$

with the solution

$$\Gamma_2 \Vdash (\mathtt{G} \circ (\mathtt{S}\ \mathtt{F})) \diamond \mathtt{N} : \mathtt{Graph}(\alpha) \to ([\alpha] \cap \mathit{TopSorted})$$

4.3 Service composition

We add a repository Γ_3 containing an interface for a task management service S, shown in Figure 11. The repository Γ_2 has been extended to Γ_4 with two components (Figure 12), one generating task graph expressing an execution plan for a set of tasks, and one that allows connecting to the service by providing a user id. The two repositories Γ_3 and Γ_4 share some semantic structures including the the domain taxonomy of Figure 10 with the sorting repository of Figure 9. Sharing semantic types leads to the possibility of composing otherwise independent components and services.

The service S has a protocol expressed in the types *SessionID*, *UserID* and *TID* – its methods must be called in sequential order, first Connect, then ReqTransaction and ending with EndTransaction. Notice that the way this is specified can be understood within the framework of coding tree automata using intersection types, as mentioned in Section 2.5 (Theorem 4). The service method ReqTransaction takes a task list (of type [Task]) and returns a transaction identifier (*TID*) which can later be used

$$
\begin{array}{lcl}
\texttt{Tr} & : & () \to \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R}) \\
\texttt{pos} & : & \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R}) \to ((\texttt{R},\texttt{R}),\texttt{R}) \\
\texttt{cdn} & : & ((\texttt{R},\texttt{R}),\texttt{R}) \to (\texttt{R},\texttt{R}) \\
\texttt{fst} & : & (\texttt{R},\texttt{R}) \to \texttt{R} \\
\texttt{snd} & : & (\texttt{R},\texttt{R}) \to \texttt{R} \\
\texttt{tmp} & : & \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R}) \to \texttt{R} \\
\texttt{cc2pl} & : & (\texttt{R},\texttt{R}) \to (\texttt{R},\texttt{R}) \\
\texttt{cl2fh} & : & \texttt{R} \to \texttt{R}
\end{array}
$$

Figure 6: Section of repository implementing a tracking service (native API)

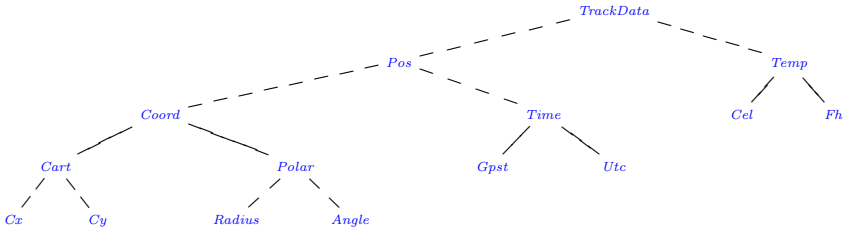


Figure 7: Semantic structures

$$
\begin{array}{lcl}
\Gamma_1 = \{ & & \\
\quad \texttt{Tr} & : & () \to \texttt{D}((\texttt{R},\texttt{R}) \cap \mathit{Cart}, \texttt{R} \cap \mathit{Gpst}, \texttt{R} \cap \mathit{Cel}) \\
\quad \texttt{pos} & : & \texttt{D}((\texttt{R},\texttt{R}) \cap \varepsilon, \texttt{R} \cap \eta, \texttt{R}) \to ((\texttt{R},\texttt{R}) \cap \varepsilon, \texttt{R} \cap \eta) \cap \mathit{Pos} \\
\quad \texttt{cdn} & : & ((\texttt{R},\texttt{R}) \cap \varepsilon, \texttt{R}) \cap \mathit{Pos} \to (\texttt{R},\texttt{R}) \cap \varepsilon \\
\quad \texttt{fst} & : & ((\texttt{R},\texttt{R}) \cap \mathit{Coord} \to \texttt{R}) \cap \\
 & & \quad (\mathit{Cart} \to \mathit{Cx}) \cap (\mathit{Polar} \to \mathit{Radius}) \\
\quad \texttt{snd} & : & ((\texttt{R},\texttt{R}) \cap \mathit{Coord} \to \texttt{R}) \cap \\
 & & \quad (\mathit{Cart} \to \mathit{Cy}) \cap (\mathit{Polar} \to \mathit{Angle}) \\
\quad \texttt{tmp} & : & \texttt{D}((\texttt{R},\texttt{R}),\texttt{R},\texttt{R} \cap \varepsilon) \to \texttt{R} \cap \varepsilon \\
\quad \texttt{cc2pl} & : & (\texttt{R},\texttt{R}) \cap \mathit{Cart} \to (\texttt{R},\texttt{R}) \cap \mathit{Polar} \\
\quad \texttt{cl2fh} & : & \texttt{R} \cap \mathit{Cel} \to \texttt{R} \cap \mathit{Fh} \\
\} & &
\end{array}
$$

Figure 8: Repository with semantic specifications

$$
\begin{array}{lcl}
\Gamma_2 = \{ & & \\
\quad \texttt{F} & : & ([\alpha] \to (\alpha \to \texttt{bool}) \to [\alpha]) \cap \mathit{Filter} \\
\quad \texttt{S} & : & (([\alpha] \to (\alpha \to \texttt{bool}) \to [\alpha]) \cap \mathit{Filter}) \to \\
 & & \quad ((\alpha \to \alpha \to \texttt{bool}) \to [\alpha] \to [\alpha]) \cap \\
 & & \quad (\mathit{TotalOrder} \to \boldsymbol{\omega} \to \mathit{Sorted}) \cap \\
 & & \quad (\mathit{PartialOrder} \to \boldsymbol{\omega} \to \mathit{TopSorted})) \\
\quad \texttt{G} & : & \texttt{Graph}(\alpha) \to ((\alpha \to \alpha \to \texttt{bool}) \cap \mathit{PartialOrder}) \\
\quad \texttt{N} & : & \texttt{Graph}(\alpha) \to [\alpha] \\
\quad \circ & : & (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma) \\
\quad \diamond & : & (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma) \\
\} & &
\end{array}
$$

Figure 9: Type environment Γ for parameterized sorting function

to retrieve a list of results (in the native form of a list of integer result codes) from `EndTransaction` (thus, one could implement `ReqTransaction` as an asynchronous method).

Connecting to the service from Γ_4 can be achieved by the composition

$$
\begin{array}{l}
\Gamma_3 \cup \Gamma_4 \Vdash \texttt{S} :: \texttt{ReqTransaction} \\
\qquad\qquad (\texttt{MyID}())(\texttt{S} :: \texttt{Connect}()) : \\
\qquad\quad ([\texttt{Task}] \cap \mathit{TopSorted}) \to (\texttt{int} \cap \mathit{TID})
\end{array}
$$

In order to increase readability in complex compositions we may write a combinatory expression $(e\ e')$ as

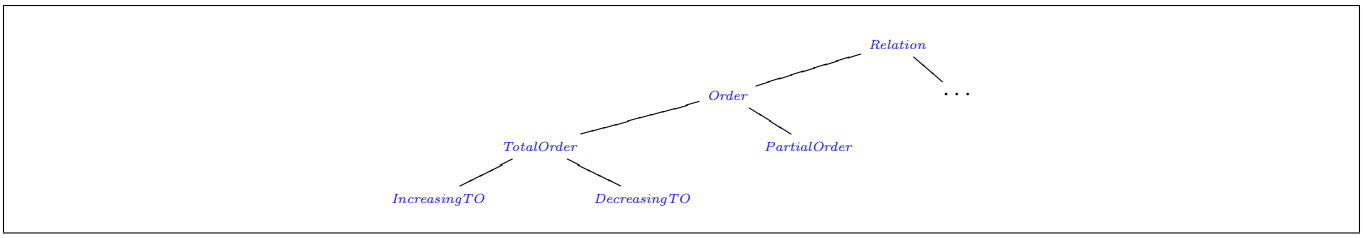


Figure 10: Semantic structures

$$
\begin{array}{lcl}
\Gamma_3 = \{ & & \\
\texttt{S} :: \texttt{Connect} & : & () \to (\texttt{int} \cap \mathit{SessionID}) \\
\texttt{S} :: \texttt{ReqTransaction} & : & (\texttt{int} \cap \mathit{UserID}) \to (\texttt{int} \cap \mathit{SessionID}) \to \\
 & & ([\texttt{Task}] \cap \mathit{TopSorted}) \to (\texttt{int} \cap \mathit{TID}) \\
\texttt{S} :: \texttt{EndTransaction} & : & (\texttt{int} \cap \mathit{TID}) \to [\texttt{int} \cap \mathit{Result}] \\
\} & &
\end{array}
$$

Figure 11: Type environment Γ for task management service

$$
\begin{array}{lcl}
\Gamma_4 = \Gamma_2 \cup \{ & & \\
\texttt{MyID} & : & () \to (\texttt{int} \cap \mathit{UserID}) \\
\texttt{GetTasks} & : & () \to \texttt{Graph}(\texttt{Task}) \\
\} & &
\end{array}
$$

Figure 12: Client environment

$$
\begin{array}{l}
\texttt{let}\ X = e \\
\ \texttt{let}\ Y = e' \\
\ \ \texttt{in}\ (X\ Y)
\end{array}
$$

We can now ask for the computation of a result list by the inhabitation goal

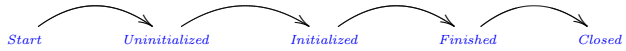
$$\Gamma_3 \cup \Gamma_4 \vdash ? : [\texttt{int} \cap \mathit{Result}]$$

which generates the solution

$$
\begin{array}{l}
\Gamma_3 \cup \Gamma_4 \vdash \texttt{let X} = \texttt{S} :: \texttt{ReqTransaction} \\
\qquad\qquad\qquad\qquad (\texttt{MyID}()) \\
\qquad\qquad\qquad\qquad\ \ (\texttt{S} :: \texttt{Connect}()) \\
\qquad\quad \texttt{let Y} = (\texttt{G} \circ (\texttt{S F})) \diamond \texttt{N} \\
\qquad\ \ \texttt{in}\ \ (\texttt{X} \circ \texttt{Y} \circ \texttt{EndTransaction})\ (\texttt{GetTasks}()) \\
\qquad : [\texttt{int} \cap \mathit{Result}]
\end{array}
$$

4.4 Protocol

In Figure 13 we have a repository with interfaces for a windowing system [9]. Semantic types are used to specify a state machine describing a protocol that determines the temporal order in which functions of the API may be called:



Consider the inhabitation goal $\Gamma_5 \vdash ? : () \cap \mathit{Closed}$. Assuming a subtyping order in the native language with

$$\texttt{layoutDesktop} \leq \texttt{layoutObj}, \texttt{layoutPDA} \leq \texttt{layoutObj}$$

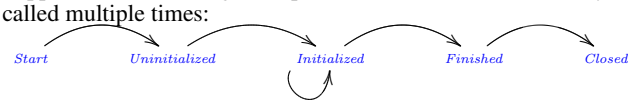
we have the following two solutions:

$$
\begin{array}{rcl}
e_1 & = & \texttt{closeWindow}(\\
 & & \quad \texttt{interact}(\\
 & & \quad\ \texttt{createControls}(\\
 & & \quad\quad \texttt{openWindow}(\\
 & & \quad\quad\ \texttt{init}()) \\
 & & \quad\quad\quad \texttt{layoutDesktopPC})))
\end{array}
$$

and

$$
\begin{array}{rcl}
e_2 & = & \texttt{closeWindow}(\\
 & & \quad \texttt{interact}(\\
 & & \quad\ \texttt{createControls}(\\
 & & \quad\quad \texttt{openWindow}(\\
 & & \quad\quad\ \texttt{init}()) \\
 & & \quad\quad\quad \texttt{layoutPDAPhone})))
\end{array}
$$

Suppose that we change the protocol so that `interact` may be called multiple times:



We now have a nondeterministic state machine, with the self-loop on the state *Initialized* and the transition from *Initialized* to *Finished*. We can easily change the specification of the combinator `interact` to implement this new protocol:

$$
\begin{array}{lcl}
\Gamma_6 = \{ & & \\
\ldots & & \\
\texttt{interact} & : & (\texttt{wndHnd} \to \texttt{wndHnd}) \cap \\
 & & (\mathit{Initialized} \to \mathit{Initialized}) \cap \\
 & & (\mathit{Initialized} \to \mathit{Finished}) \\
\ldots & & \\
\} & &
\end{array}
$$

The inhabitation algorithm will now generate infinitely many solutions to the goal $\Gamma_5 \vdash ? : () \cap \mathit{Closed}$, since there may now be 1 or more calls to `interact`. The **CLS** algorithm can discover such loops and can be instructed to deal with them in different ways (loops are discovered proof-theoretically: a proof of τ containing a subproof of τ is circular). Usually, we report only non-circular proofs.

For the purpose of understanding and representing the solution space to an inhabitation problem, the connection to the theory of tree automata mentioned in Section 2.5 can be useful. It follows from the results mentioned there that we can regard the inhabita-

$$
\begin{array}{lcl}
\Gamma_5 = \{ & & \\
\quad \texttt{init} & : & () \to (() \cap \mathit{Start}) \\
\quad \texttt{layoutDesktopPC} & : & \texttt{layoutDesktop} \\
\quad \texttt{layoutPDAPhone} & : & \texttt{layoutPDA} \\
\quad \texttt{openWindow} & : & (() \to \texttt{wndHnd}) \cap (\mathit{Start} \to \mathit{Uninitialized}) \\
\quad \texttt{createControls} & : & (\texttt{wndHnd} \cap \mathit{Uninitialized}) \to \texttt{layoutObj} \to \\
 & & \quad (\texttt{wndHnd} \cap \mathit{Initialized}) \\
\quad \texttt{interact} & : & (\texttt{wndHnd} \to \texttt{wndHnd}) \cap (\mathit{Initialized} \to \mathit{Finished}) \\
\quad \texttt{closeWindow} & : & (\texttt{wndHnd} \to ()) \cap (\mathit{Finished} \to \mathit{Closed}) \\
\} & &
\end{array}
$$

Figure 13: Type environment Γ for protocol-based synthesis in abstract windowing system

tion algorithm as a tree recognizer and the solution space to an inhabitation question as a tree language

$$\mathbf{L}(\Gamma, \tau) = \{e \mid \Gamma \vdash e : \tau\}$$

In the case of a regular set of inhabitants we could choose to present the infinite set of solutions as a regular tree expression. For our previous inhabitation problem we could present the solutions by the expression

$$
\begin{array}{l}
\mathbf{L}(\Gamma_6, () \cap \mathit{Closed}) = \\
\quad \texttt{closeWindow}(\\
\quad\quad \texttt{interact}^{+}(\\
\quad\quad\quad \texttt{createControls}(\\
\quad\quad\quad\quad \texttt{openWindow}(\\
\quad\quad\quad\quad\quad \texttt{init}()) \\
\quad\quad\quad\quad\quad\quad (\texttt{layoutDesktopPC}{+}\texttt{layoutPDAPhone}))))
\end{array}
$$

4.5 Logic

We can go beyond purely taxonomic relations between semantic concepts by introducing combinator types that express more general logical relations between such concepts (leading to what is sometimes referred to as "ontologies"). For example, referring again to the taxonomy of Figure 10, suppose that we wish to introduce the concept of relation reversal, Rev, so that for a semantic type R denoting a class of binary relations, the semantic type $\mathit{Rev}(R)$ denotes the class of binary relations that are reversals of R. Technically, we could define the operation on types

$$\mathit{Rev}(R) = \{r^{-1} \mid r \in R\}$$

where $r^{-1} = \{(y, x) \mid (x, y) \in r\}$. This definition may be given outside the type system as a specification of the intended semantics of the operation Rev. But we may wish to reflect how the operation Rev acts on types in the semantic type system. For example, on the types shown in Figure 10 we could have the properties

$$\mathit{Rev}(\mathit{Relation.DecreasingTO}) = \mathit{Relation.IncreasingTO}$$

and

$$\mathit{Rev}(\mathit{Relation.IncreasingTO}) = \mathit{Relation.DecreasingTO}$$

and we might be interested in using such relations to control inhabitation.

Consider the repository Γ_7 shown in Figure 14 which extends the repository Γ_2 (Figure 9). The combinator $\Phi_{\texttt{Rev}}$ can be seen as a noop (the identity function $\lambda x.x$) whose only function is to express a logical fact. The type of $\Phi_{\texttt{Rev}}$ expresses the abstract behaviour of the semantic operation Rev on the types

$$\mathit{Relation.IncreasingTO}, \mathit{Relation.DecreasingTO}$$

– the reversal of an increasing total order is a decreasing total order, and vice versa. The underlying identity type $\alpha \to \alpha$ captures the fact that $\Phi_{\texttt{Rev}}$ is a purely *logical combinator* with no operational significance. Generally, purely logical facts φ can be expressed by combinators $\Phi : (\alpha \to \alpha) \cap \varphi$. We can now use this representation of the semantic action Rev to express the behaviour of the combinator $\texttt{swap}$ (with the intended operational semantics $\lambda f.\lambda x.\lambda y.fyx$). The type of $\texttt{swap}$ implies that, when applied to a binary relation, it performs the operation of relational reversal ($\texttt{swap}$ might have other interesting properties besides).

We can now use the parameterized sorting function on the reversal of a given order relation. The inhabitation goal

$$\Gamma_7 \Vdash ? : [\texttt{int}] \to ([\texttt{int}] \cap \mathit{Sorted}(\mathit{DecreasingTO}))$$

asks for a program that can sort integers in decreasing order. It leads to synthesis of the following composition:

$$
\begin{array}{l}
\Gamma_7 \Vdash \texttt{S F} (\Phi_{\texttt{Rev}} (\texttt{swap} \leq_{\mathbb{N}})) : \\
\qquad [\texttt{int}] \to ([\texttt{int}] \cap \mathit{Sorted}(\mathit{DecreasingTO}))
\end{array}
$$

5. Conclusion and future work

We have attempted to outline the logical foundations of a new methodology that could be used for automatic synthesis of component composition based on relativized inhabitation in combinatory logic. We have given a number of illustrative examples which, even though small and simple, hopefully indicate that the research programme outline here might be interesting to pursue further.

There remains a large number of research questions that need to be pursued, including the following.

It would be desirable to obtain a further stratification of the logical system by which we can distinguish between composition-time combinators and run-time (native) combinators. This would enable the framework to use meta-syntactic composition-time operators which can be executed, at composition time, to generate native compositions (for example, using native code templates and substitutions into such). It would support a staged computation model (composition-time vs. run-time execution). Also, it would make it possible to operate with native languages that are radically different from the language within which compositions are computed (for example, process languages). We are working on an extension to the framework using modal types to accomplish this.

A very large set of questions pertain to optimization of the inhabitation algorithm, and we are currently working on several optimizations to the inhabitation algorithm in **CLS**.

Finally, of foremost importance is continued experimental work to explore the specification and "programming" model of combinatory logic inhabitation and to demonstrate the possibilies and limitations of combinatory logic synthesis.

Acknowledgments

I am most grateful to my collaborators on combinatory logic and synthesis, Boris Düdder, Moritz Martens and Paweł Urzyczyn, and to Roger Hindley for very helpful discussions about combinatory and propositional logic.

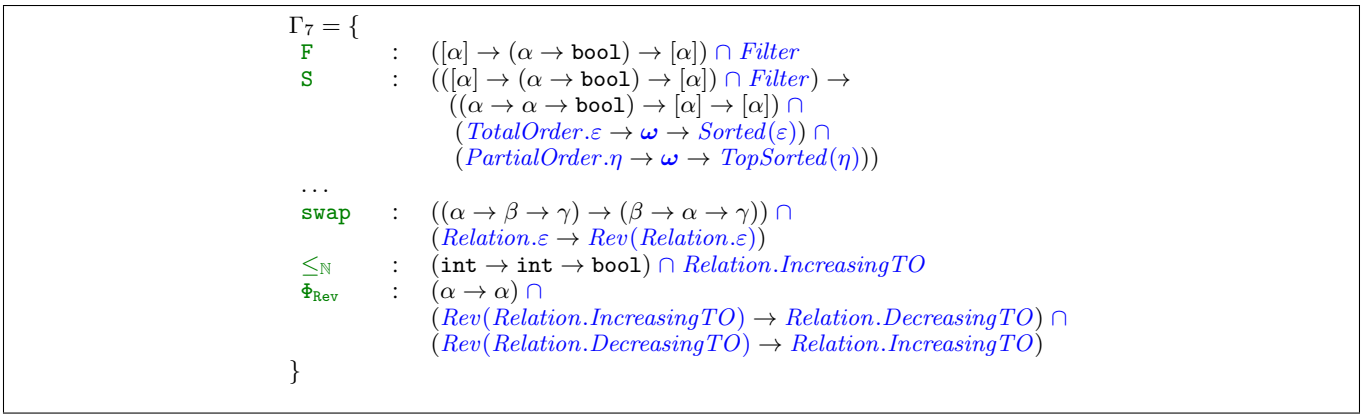


Figure 14: Parameterized sorting function with logical combinator

References

[1] K. Bimbó and J. Dunn. On the decidability of implicational ticket entailment. *Journal of Symbolic Logic*. To appear.

[2] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[3] M. Coppo and M. Dezani-Ciancaglini. An extension of basic functionality theory for lambda-calculus. *Notre Dame Journal of Formal Logic*, 21:685–693, 1980.

[4] M. Davis. *Computability and Unsolvability*. McGraw-Hill Book Co., 1958.

[5] M. Dezani-Ciancaglini and R. Hindley. Intersection types for combinatory logic. *Theoretical Computer Science*, 100(2):303–324, 1992.

[6] B. Düdder and M. Martens. **CLS**, combinatory logic synthesizer. Website `http://ls14-www.cs.uni-dortmund.de/index.php/CLS`.

[7] B. Düdder, O. Garbe, M. Martens, J. Rehof, and P. Urzyczyn. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI Synthesis. In *Proceedings of ITRS'12*, 2012.

[8] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn. Bounded Combinatory Logic. In *Computer Science Logic (CSL'12)*, volume 16 of *LIPIcs*, pages 243–258. Leibniz-Zentrum fuer Informatik, 2012.

[9] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Synthesis. Technical Report 842, Faculty of Computer Science (TU Dortmund), 2012. `http://ls14-www.cs.tu-dortmund.de/index.php/Jakob_Rehof_Publications#Technical_Reports`.

[10] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 268–277. ACM, 1991.

[11] M. D. Gladstone. Some ways of constructing a propositional calculus of any required degree of unsolvability. *Transactions of the American Mathematical Society*, 118:195–210, 1965.

[12] C. Haack, B. Howard, A. Stoughton, and J. B. Wells. Fully automatic adaptation of software components based on semantic specifications. In *AMAST'02*, volume 2422 of *LNCS*, pages 83–98. Springer, 2002.

[13] J. R. Hindley and J. P. Seldin. *Lambda-calculus and Combinators, an Introduction*. Cambridge University Press, 2008.

[14] L. Linial and E. L. Post. Recursive unsolvability of the deducibility, Tarski's completeness and independence of axioms problems of propositional calculus. *Bulletin of the American Mathematical Society*, 55:50, 1949.

[15] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2): 125–157, 1991.

[16] V. Padovani. Ticket entailment is decidable. *CoRR abs/1106.1875*, 2011.

[17] G. Pottinger. A type assignment for the strongly normalizable lambda-terms. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.

[18] J. Rehof and P. Urzyczyn. Finite combinatory logic with intersection types. In *TLCA*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011.

[19] J. Rehof and P. Urzyczyn. Finite combinatory logic with intersection types. Technical Report 834, Dept. of Computer Science, Technical University of Dortmund, `http://ls14-www.cs.tu-dortmund.de/index.php/Datei:TR-834.pdf`, 2011.

[20] S. Salvati. Recognizability in the simply typed lambda-calculus. In H. Ono, M. Kanazawa, and R. J. G. B. de Queiroz, editors, *WoLLIC*, volume 5514 of *LNCS*, pages 48–60. Springer, 2009.

[21] S. Salvati, G. Manzonetto, M. Gehrke, and H. Barendregt. Loader and Urzyczyn are logically related. In *ICALP 12, Automata, Languages, and Programming - 39th International Colloquium, Warwick, UK*, volume 7392 of *LNCS*, pages 364–376. Springer, 2012.

[22] W. E. Singletary. Recursive unsolvability of a complex of problems proposed by Post. *Journal of the Faculty of Science, University of Tokyo*, 14:25–58, 1967.

[23] W. E. Singletary. Many-one degrees associated with partial propositional calculi. *Notre Dame Journal of Formal Logic*, XV(2):335–343, 1974.

[24] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.

[25] P. Trigg, J. Hindley, and M. Bunder. Combinatory abstraction using $\mathbf{B}, \mathbf{B}'$ and friends. *Theoretical Computer Science*, 135:405–422, 1994.

[26] A. Urquhardt. The undecidability of entailment and relevant implication. *The Journal of Symbolic Logic*, 49(4), 1984.

[27] P. Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

[28] M. Yntema. A detailed argument for the Post-Linial theorems. *Notre dame Journal of Formal Logic*, V(1), 1964.