# Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Composition Synthesis

Boris Düdder   Oliver Garbe   Moritz Martens   Jakob Rehof
Faculty of Computer Science
Technical University of Dortmund
Dortmund, Germany
{boris.duedder, oliver.garbe, moritz.martens, jakob.rehof}@cs.tu-dortmund.de

Paweł Urzyczyn
Institute of Informatics
University of Warsaw
Warsaw, Poland
urzy@mimuw.edu.pl

We describe ongoing work on a framework for automatic composition synthesis from a repository of software components. This work is based on combinatory logic with intersection types. The idea is that components are modeled as typed combinators, and an algorithm for inhabitation — is there a combinatory term $e$ with type $\tau$ relative to an environment $\Gamma$? — can be used to synthesize compositions. Here, $\Gamma$ represents the repository in the form of typed combinators, $\tau$ specifies the synthesis goal, and $e$ is the synthesized program. We illustrate our approach by examples, including an application to synthesis from GUI-components.

## 1 Introduction

In this paper we describe ongoing work to construct and apply a framework for automatic composition synthesis from software repositories, based on inhabitation in combinatory logic with intersection types. We describe the basic idea of type-based synthesis using bounded combinatory logic with intersection types and illustrate an application of the framework to the synthesis of graphical user interfaces (GUIs). Although our framework is under development and hence results in applications to synthesis are still preliminary, we hope to illustrate an interesting new approach to type-based synthesis from component repositories.

In a recent series of papers [18, 19, 7] we have laid the theoretical foundations for understanding algorithmics and complexity of decidable inhabitation in subsystems of the intersection type system [4]. In contrast to standard combinatory logic where a fixed basis of combinators is usually considered, the inhabitation problem considered here is *relativized* to an arbitrary environment $\Gamma$ given as part of the input. This problem is undecidable for combinatory logic, even in simple types, see [7]. We have introduced finite and bounded combinatory logic with intersection types in [18, 7] as a possible foundation for type-based composition synthesis. *Finite combinatory logic* (abbreviated FCL) [18] arises from combinatory logic by restricting combinator types to be monomorphic, and *k-bounded combinatory logic* (abbreviated $\text{BCL}_k$) [7] is obtained by imposing the bound $k$ on the depth of types that can be used to instantiate polymorphic combinator types. It was shown that relativized inhabitation in finite combinatory logic is EXPTIME-complete [18], and that $k$-bounded combinatory logic forms an infinite hierarchy depending on $k$, inhabitation being $(k+2)$-EXPTIME-complete for each $k \geq 0$. In this paper, we stay within the lowest level of the hierarchy, BCL$_0$. We note that, already at this level, we have a framework for 2-EXPTIME-complete synthesis problems, equivalent in complexity to other known synthesis frameworks (e.g., variants of temporal logic and propositional dynamic logic).

In positing bounded combinatory logic as a foundation for composition synthesis, we consider the *inhabitation problem*: Given an environment $\Gamma$ of typed combinators and a type $\tau$, does there exist a combinatory term $e$ such that $\Gamma \vdash e : \tau$? For applications in synthesis, we consider $\Gamma$ as a repository of components represented only by their names (combinators) and their types (intersection types), and $\tau$ is seen as the specification of a synthesis goal. An inhabitant $e$ is a program obtained by applicative combination of components in $\Gamma$. The inhabitant $e$ is automatically constructed (synthesized) by the inhabitation algorithm. For applications to synthesis, where the repository $\Gamma$ may vary, the relativized inhabitation problem is the natural model.

## 2 Inhabitation in Finite and Bounded Combinatory Logic

We state the necessary notions and definitions for *finite and bounded combinatory logic with intersection types and subtyping* [18, 7]. We consider *applicative terms* ranged over by $e$, etc. and defined as

$$e ::= x \mid (e\, e'),$$

where $x$, $y$ and $z$ range over a denumerable set of *variables* also called *combinators*. As usual, we take application of terms to be left-associative. Under these premises any applicative term can be uniquely written as $xe_1 \dots e_n$ for some $n \geq 0$. Sometimes we may also write $x(e_1, \dots, e_n)$ instead of $xe_1 \dots e_n$. *Types*, ranged over by $\tau$, $\sigma$, etc. are defined by

$$\tau ::= a \mid \tau \to \tau \mid \tau \cap \tau$$

where $a, b, c, \dots$ range over *atoms* comprising *type constants* from a finite set $\mathbb{A} \uplus \{\omega\}$ and *type variables* from a disjoint denumerable set $\mathbb{V}$ ranged over by $\alpha, \beta, \gamma, \dots$ The constant $\omega$ is the top-element with regard to the subtyping relation $\leq$ defined below. We denote the set of all types by $\mathbb{T}$. As usual, intersections are idempotent, commutative, and associative. Notationally, we take the type constructor $\to$ to be right-associative. A type $\tau \cap \sigma$ is called an *intersection type* or *intersection* [16, 4] and is said to have $\tau$ and $\sigma$ as *components*. We sometimes write $\bigcap_{i=1}^{n} \tau_i$ for an intersection with $n \geq 1$ components. Intersection types come with a natural notion of subtyping, as defined in [4]. The subtyping relation, denoted by $\leq$, is the least preorder (reflexive and transitive relation) on $\mathbb{T}$ satisfying the following conditions:

$$\sigma \leq \omega, \quad \omega \leq \omega \to \omega, \quad \sigma \cap \tau \leq \sigma, \quad \sigma \cap \tau \leq \tau, \quad \sigma \leq \sigma \cap \sigma;$$
$$(\sigma \to \tau) \cap (\sigma \to \rho) \leq \sigma \to \tau \cap \rho;$$
$$\text{If } \sigma \leq \sigma' \text{ and } \tau \leq \tau' \text{ then } \sigma \cap \tau \leq \sigma' \cap \tau' \text{ and } \sigma' \to \tau \leq \sigma \to \tau'.$$

Subtyping is used in the systems of [18, 7], and it is decidable in polynomial time [18]. We say that two types $\tau$ and $\sigma$ are *equivalent* if and only if $\tau \leq \sigma$ and $\sigma \leq \tau$.

If $\tau = \tau_1 \to \cdots \to \tau_n \to \sigma$ we write $\sigma = tgt_n(\tau)$ and $\tau_i = arg_i(\tau)$ for $i \leq n$ and we say that $\sigma$ is a *target* type of $\tau$ and $\tau_i$ are *argument* types of $\tau$. A type of the form $\tau = \tau_1 \to \cdots \to \tau_n \to a$ with $a \neq \omega$ an atom is called a *path* of length $n$. A type is *organized* if it is an intersection of paths. For every type $\tau$ there is an equivalent organized type $\bar{\tau}$ that is computable in polynomial time [19]. Therefore, in the following we assume all types to be organized. For $\sigma \in \mathbb{T}$ we denote by $\mathbb{P}_n(\sigma)$ the set of all paths of length greater than or equal to $n$ in $\sigma$ and by $\parallel \sigma \parallel$ the path length of $\sigma$ which is defined to be the maximal length of a path in $\sigma$. Define the set $\mathbb{T}_0$ of *level* 0 *types* by $\mathbb{T}_0 = \{\bigcap_{i \in I} a_i \mid a_i$ an atom, $I$ a finite index set$\}$. Thus, level 0 types comprise of atoms and intersections of such. We write $\mathbb{T}_0(\Gamma, \tau)$ to denote the set of level 0 types with atoms from $\Gamma$ and $\tau$. Note that $\omega$ is also contained in $\mathbb{T}_0(\Gamma, \tau)$. A *substitution* is a function $S : \mathbb{V} \to \mathbb{T}_0$ such that $S$ is the identity everywhere but on a finite subset of $\mathbb{V}$. We tacitly lift $S$ to a function on types, $S : \mathbb{T} \to \mathbb{T}$, by homomorphic extension. A *type environment* $\Gamma$ is a finite set of type assumptions of the form $x : \tau$.

$$\frac{[S : \mathbb{V} \to \mathbb{T}_0]}{\Gamma, x : \tau \vdash x : S(\tau)}(\text{var}) \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\, e') : \tau'}(\to\text{E})$$

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

**Figure 1:** $\text{BCL}_0(\cap, \leq)$

The type rules for 0-*bounded combinatory logic with intersection types and subtyping*, denoted $\text{BCL}_0(\cap, \leq)$ or simply $\text{BCL}_0$, as presented in [7], are given in Figure 1. The bound 0 is enforced by the fact that only substitutions $S$ mapping type variables to level 0 types in $\mathbb{T}_0$ are allowed in rule (var). In effect, $\text{BCL}_0$ allows a limited form of polymorphism of combinators in $\Gamma$, where type variables can be instantiated with atomic types or intersections of such. *Finite combinatory logic with intersection types and subtyping*, denoted $\text{FCL}(\cap, \leq)$, as presented in [18], is the monomorphic restriction of $\text{BCL}_0(\cap, \leq)$ where the substitutions $S$ in rule (var) of Figure 1 are required to be the identity. Hence, rule (var) simplifies to the axiom $\Gamma, x : \tau \vdash x : \tau$.

We consider the *relativized inhabitation problem*:

*Given an environment $\Gamma$ and a type $\tau$, does there exist an applicative term e such that $\Gamma \vdash e : \tau$?*

We sometimes write $\Gamma \vdash ? : \tau$ to indicate an inhabitation goal. In [18] it is shown that deciding inhabitation in $\text{FCL}(\cap, \leq)$ is EXPTIME-complete. The lower bound is by reduction from the intersection non-emptiness problem for finite bottom-up tree automata, and the upper-bound is by constructing a polynomial space bounded alternating Turing machine (ATM) [3]. In [7] it is shown that $k$-bounded combinatory logic (where substitutions are allowed in rule (var) mapping type variables to types of depth at most $k$) is $(k+2)$-EXPTIME-complete for every $k \geq 0$, and hence the lowest level of the bounded hierarchy $\text{BCL}_0(\cap, \leq)$ is 2-EXPTIME-complete. The lower bound for $\text{BCL}_0$ is by reduction from acceptance of an exponential space bounded ATM.

The 2-EXPTIME (alternating exponential space) algorithm is shown in Figure 2. In Figure 2 we use shorthand notation for ATM-instruction sequences starting from existential states (CHOOSE...) and instruction sequences starting from universal states (FORALL$(i = 1 \ldots n)\, s_i$). A command of the form CHOOSE $x \in P$ branches from an existential state to successor states in which $x$ gets assigned distinct elements of $P$. A command of the form FORALL$(i = 1 \ldots n)\, s_i$ branches from a universal state to successor states from which each instruction sequence $s_i$ is executed. The machine is exponential space bounded, because the set of substitutions $Var(\Gamma, \tau) \to \mathbb{T}_0(\Gamma, \tau)$ is exponentially bounded. We refer to [7] for further details.

```
          Input :  $\Gamma, \tau$

1      // loop
2         CHOOSE $(x : \sigma) \in \Gamma$;
3         $\sigma' := \bigcap\{S(\sigma) \mid S : Var(\Gamma,\tau) \to \mathbb{T}_0(\Gamma,\tau)\}$;
4         CHOOSE $n \in \{0, \ldots, \| \sigma' \|\}$;
5         CHOOSE $P \subseteq \mathbb{P}_n(\sigma')$;

6         IF $(\bigcap_{\pi \in P} tgt_n(\pi) \leq \tau)$ THEN
7            IF $(n = 0)$ THEN ACCEPT;
8            ELSE
9               FORALL$(i = 1 \ldots n)$
10                    $\tau := \bigcap_{\pi \in P} arg_i(\pi)$;
11              GOTO LINE 2;
12        ELSE REJECT;
```

**Figure 2:** Alternating Turing machine $\mathscr{M}$ deciding inhabitation for $\mathrm{BCL}_0(\cap, \leq)$

## 3 Synthesis from Component Repositories

In this section we briefly summarize some main points of our methodology for composition synthesis, and we illustrate some of the main principles by an idealized example (the reader might want to take a preliminary look at the example in Section 3.2 first). We should emphasize that we only aim at an intuitive presentation of the general idea in broad outline, and there are many further aspects to our proposed method that cannot be discussed here for space reasons. The paper [17] contains a more theoretical account of the methodology and has further examples.

### 3.1 Basic principles

#### Semantic specification

It is well known that intersection types can be used to specify deep semantic properties in the $\lambda$-calculus. The system characterizes the strongly normalizing terms [16, 4], the inhabitation problem is closely related to the $\lambda$-definability problem [21, 22], and our work on bounded combinatory logic [18, 7] shows that $k$-bounded inhabitation can code any exponential level of space bounded alternating Turing machines, depending on $k$. Many existing applications of intersection types testify to their expressive power in various applications. Moreover, it is simple to prove but interesting to note that we can specify any given term $e$ uniquely: there is an environment $\Gamma_e$ and a type $\tau_e$ such that $e$ is the unique term with $\Gamma_e \vdash e : \tau_e$ (see [18]).

#### A type-based, taxonomic approach

It is a possible advantage of the type-based approach advocated here (in comparison to, e.g., approaches based on temporal logic) that types can be naturally associated with code, because application programming interfaces (APIs) already have types. In our applications, we think of intersection types as hosting, in principle, 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 (domain ontology). For example, we might consider a specification

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

where native types ($\mathtt{real}$, $\mathtt{real} \times \mathtt{real}, \ldots$) are qualified, using intersections with semantic types (in the example, $\mathit{Cart}, \mathit{Pol}, \mathit{Iso}$) expressing (relative to a given conceptual taxonomy) interesting domain-specific properties of the function (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., $\mathit{Cart}, \mathit{Pol}$ are features of coordinates, $\mathit{Iso}$ is a feature of functions) whose elements can be mapped onto the native API using intersections, at any level of the type structure.

## Level 0-bounded polymorphism

The main difference between FCL and BCL$_0$ lies in succinctness of BCL$_0$. For example, consider that we can represent any finite function $f : A \to B$ as an intersection type $\tau_f = \bigcap_{a \in A} a \to f(a)$, where elements of $A$ and $B$ are type constants. Suppose we have combinators $(F_i : \tau_{f_i}) \in \Gamma$, and we want to synthesize compositions of such functions represented as types (in some of our applications they could, for example, be refinement types [9]). We might want to introduce composition combinators of arbitrary arity, say $g : (A \to A)^n \to (A \to A)$. In the monomorphic system, a function table for $g$ would be exponentially large in $n$. The single declaration $G : (\alpha_0 \to \alpha_1) \to (\alpha_1 \to \alpha_2) \to \cdots \to (\alpha_{n-1} \to \alpha_n) \to (\alpha_0 \to \alpha_n)$ in $\Gamma$ can be used in BCL$_0$ to represent $g$. Through level 0 polymorphism, the action of $g$ is thereby fully specified.

Generally, the level BCL$_0$ is already very expressive (the inhabitation problem for BCL$_0$ is equivalent to the acceptance problem for alternating exponential space bounded Turing machines, hence 2-EXPTIME complete [7]). The question of expressive power in practice (how easy or hard it is to specify given classes of practical problems) is harder to answer in general and at this stage of our experience. So far, we have found the formalism of intersection types to be very versatile. More experimental work is needed, however. Another question in this context that would be interesting to consider in future work is the connection to temporal logic synthesis problems (many of which are also 2-EXPTIME complete).

## Typed repositories as composition logic programs

When considering the inhabitation problem $\Gamma \vdash ? : \tau$ as a foundation for synthesis, it may be useful to think of $\Gamma$ as a form of generalized logic program, broadly speaking, along the lines of the idea of proof theoretical logic programming languages proposed by Miller et al. [15]. Under this viewpoint, solving the inhabitation problem $\Gamma \vdash ? : \tau$ means evaluating the program $\Gamma$ against the goal $\tau$: each typed combinator $F : \sigma$ in $\Gamma$ names a single logical "rule" (type $\sigma$) in an implicational logic, and the repository $\Gamma$ (a collection of such rules) constitutes a logic "program", which, when given a goal formula $\tau$ (type inhabitation target), determines the set of solutions (the set of inhabitants). In other words, the "rule" (type) of a combinator expresses how the combinator composes with other combinators and how its use contributes to goal resolution in the wider "program" (repository, $\Gamma$). Indeed, we can view the search procedure of the inhabitation algorithm shown in Figure 2 as an operational semantics for such programs.

## 3.2 An example repository

We consider a simple, idealized example to illustrate some key ideas in synthesis based on bounded combinatory logic. Consider the section of a repository of functions shown in Figure 3, where the native API of a tracking service is given as a type environment consisting of bindings `f` : `T` where `f` is the name of a function (combinator), and `T` is a native (implementation) type. We can think of the native repository as a Java API, for example, 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 data structure 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 and that time. 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 data structure `D`, with `pos` returning the position (coordinate and time), `cdn` projects the coordinates from the components of a position, `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.

| | | |
|---|---|---|
| `Tr` | : | $() \to \mathtt{D((R,R),R,R)}$ |
| `pos` | : | $\mathtt{D((R,R),R,R)} \to \mathtt{((R,R),R)}$ |
| `cdn` | : | $\mathtt{((R,R),R)} \to \mathtt{(R,R)}$ |
| `fst` | : | $\mathtt{(R,R)} \to \mathtt{R}$ |
| `snd` | : | $\mathtt{(R,R)} \to \mathtt{R}$ |
| `tmp` | : | $\mathtt{D((R,R),R,R)} \to \mathtt{R}$ |
| `cc2pl` | : | $\mathtt{(R,R)} \to \mathtt{(R,R)}$ |
| `cl2fh` | : | $\mathtt{R} \to \mathtt{R}$ |

**Figure 3:** Section of repository implementing a tracking service (native API)

Now, the problem with the standard, native API shown in Figure 3 is that it does not express any of the semantics of its intended use as described above. The basic idea behind combinatory logic synthesis with intersection types is that we can *use intersection types to superimpose conceptual structure onto the native API in order to express semantic properties*. In order to do so, we must first specify a suitable conceptual structure to capture the intended semantics. Figure 4 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.

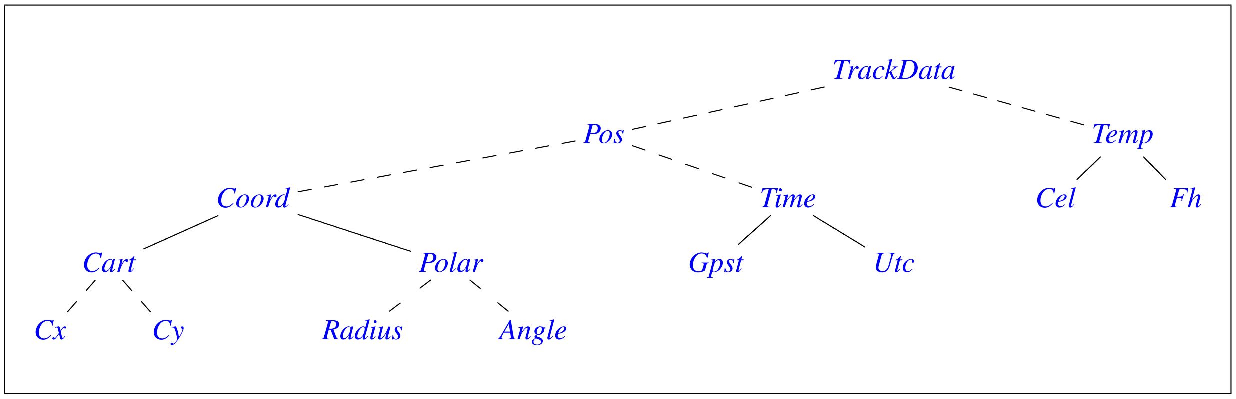


**Figure 4:** Semantic structures

$$
\begin{array}{lll}
\Gamma = \{ & & \\
\texttt{Tr} & : & () \to \mathtt{D}((\mathtt{R},\mathtt{R}) \cap \mathit{Cart}, \mathtt{R} \cap \mathit{Gpst}, \mathtt{R} \cap \mathit{Cel}), \\
\texttt{pos} & : & \mathtt{D}((\mathtt{R},\mathtt{R}) \cap \alpha, \mathtt{R} \cap \beta, \mathtt{R}) \to ((\mathtt{R},\mathtt{R}) \cap \alpha, \mathtt{R} \cap \beta) \cap \mathit{Pos}, \\
\texttt{cdn} & : & ((\mathtt{R},\mathtt{R}) \cap \alpha, \mathtt{R}) \cap \mathit{Pos} \to (\mathtt{R},\mathtt{R}) \cap \alpha, \\
\texttt{fst} & : & ((\mathtt{R},\mathtt{R}) \cap \mathit{Coord} \to \mathtt{R}) \cap \\
 & & (\mathit{Cart} \to \mathit{Cx}) \cap (\mathit{Polar} \to \mathit{Radius}), \\
\texttt{snd} & : & ((\mathtt{R},\mathtt{R}) \cap \mathit{Coord} \to \mathtt{R}) \cap \\
 & & (\mathit{Cart} \to \mathit{Cy}) \cap (\mathit{Polar} \to \mathit{Angle}), \\
\texttt{tmp} & : & \mathtt{D}((\mathtt{R},\mathtt{R}), \mathtt{R}, \mathtt{R} \cap \gamma) \to \mathtt{R} \cap \gamma, \\
\texttt{cc2pl} & : & (\mathtt{R},\mathtt{R}) \cap \mathit{Cart} \to (\mathtt{R},\mathtt{R}) \cap \mathit{Polar}, \\
\texttt{cl2fh} & : & \mathtt{R} \cap \mathit{Cel} \to \mathtt{R} \cap \mathit{Fh} \\
\} & &
\end{array}
$$

**Figure 5:** Repository with semantic specifications

In Figure 5 we show the repository of Figure 3 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 coordinates are Cartesian, time is GPS, and temperature is Celsius. Level 0 polymorphic type variables ($\alpha, \beta, \gamma$) 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 (the variable $\alpha$ standing for the semantic qualification of the coordinate component, the variable $\beta$ 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 [9] 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 5 considered as a combinatory type environment $\Gamma$ 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 $\Gamma \vdash ? : \mathit{Radius}$. Sending this question to our inhabitation algorithm gives back the (in this case unique) solution

$$\Gamma \vdash \texttt{fst}\ (\texttt{cc2pl}\ (\texttt{cdn}\ (\texttt{pos}\ \texttt{Tr}()))) : \mathit{Radius}$$

We should note that the concept of a “current radius” is relative to the given repository. If, for instance, the repository were extended with a further component $\texttt{origin} : (\texttt{R},\texttt{R}) \cap \mathit{Cart}$, then we should have the additional solution $\texttt{fst}\ (\texttt{cc2pl}\ \texttt{origin})$ to the query above, which is not the current radius but *a* radius. To express the idea of a current radius in the extended repository one would have to extend the specifications. Generally, it can be shown [18] that any given combinatory term can always be specified uniquely, providing suitable specifications in the repository. But in many practical situations it is to be expected that queries could yield multiple solutions, which might in some situations be of practical value and in others the opposite. It is therefore of some interest (also at the time of design of a semantic repository) to be able to discover and present the structure of solution spaces to queries. As shown in [18] this can be done, decision procedures for emptiness, uniqueness and finiteness of solutions are given there. Moreover, one can consider compactly representing larger (or infinite) sets of solutions, as is briefly discussed in [17].

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 Applications to GUI synthesis

In this section we focus on the application of inhabitation in bounded combinatory logic in a larger software engineering context. We illustrate how the inhabitation algorithm is integrated into a framework for synthesis from a repository. We have applied the framework to various application domains, including synthesis of control instructions for Lego NXT robots, concurrent workflow synthesis, protocol-based program synthesis, and graphical user interfaces. Below, we will discuss the two last mentioned applications in more detail.

### 4.1 Protocol-Based Synthesis for Windowing Systems

Based on protocols we use inhabitation to synthesize programs where the protocols determine the intended program behavior. We give a proof-of-concept example. It illustrates how intersection types can be used to connect different types — native types and semantic types — such that data constraints are satisfied whereas semantic types are used to control the result of the synthesis. Figure 6 shows a type environment $\Gamma$ which models a GUI programming scenario for an abstract windowing system. Further, we define the subtyping relations $\texttt{layoutDesktop} \leq \texttt{layoutObj}$ and $\texttt{layoutPDA} \leq \texttt{layoutObj}$. In this scenario we aim to synthesize a program which opens a window, populates it with GUI controls, allows a user interaction, and closes it. The typical data types like `wndHnd` (window handle) model API data types. Semantic types like *initialized* express the current state of the protocol. Type inhabitation can now be used to synthesize the program described above by asking the inhabitation question $\Gamma \vdash ? : \mathit{closed}$. The inhabitants

$$e_1 := \texttt{closeWindow}(\texttt{interact}(\texttt{createControls}(\texttt{openWindow}(\texttt{init}), \texttt{layoutDesktopPC})))$$

$$e_2 := \texttt{closeWindow}(\texttt{interact}(\texttt{createControls}(\texttt{openWindow}(\texttt{init}), \texttt{layoutPDAPhone})))$$

share the same type *closed* because both `layoutDesktop` and `layoutPDA` are subtypes of `layoutObj`. Both terms $e_1$ and $e_2$ can be interpreted or compiled to realize the intended behavior. These terms are type correct and in addition semantically correct (cf. Haack et al. [11]), because all specification axioms defined by the semantic types are satisfied.

$$
\begin{aligned}
\Gamma = \{ \quad \mathtt{init} &: \mathit{start}, \\
\mathtt{layoutDesktopPC} &: \mathtt{layoutDesktop}, \\
\mathtt{layoutPDAPhone} &: \mathtt{layoutPDA}, \\
\mathtt{openWindow} &: \mathit{start} \to \mathtt{wndHnd} \cap \mathit{uninitialized}, \\
\mathtt{createControls} &: \mathtt{wndHnd} \cap \mathit{uninitialized} \to \mathtt{layoutObj} \to \mathtt{wndHnd} \cap \mathit{initialized}, \\
\mathtt{interact} &: \mathtt{wndHnd} \cap \mathit{initialized} \to \mathtt{wndHnd} \cap \mathit{finished}, \\
\mathtt{closeWindow} &: \mathtt{wndHnd} \cap \mathit{finished} \to \mathit{closed} \}
\end{aligned}
$$

**Figure 6:** Type environment $\Gamma$ for protocol-based synthesis in abstract windowing system

## 4.2 GUI Synthesis from a Repository

We describe the application of our inhabitation algorithm in a larger framework for component-based GUI-development [12], thereby enabling automatic synthesis of GUI-applications from a repository of components. The main point we wish to illustrate is the integration of inhabitation in a more complex software synthesis framework, where combinators may represent a variety of objects, including code templates or abstract structures representing GUI components, into which other components need to be substituted in order to build the desired software application. The main formalism used by [12] to describe the GUI to be synthesized are abstract interaction nets (AINs). An AIN is an extended Petri net and is used to describe complex interaction patterns occurring in the GUI. The places of an AIN represent the objects of the GUI that are involved in the pattern described. The presence of tokens in a place represents the fact that the corresponding GUI-object is active. The transitions of an AIN represent interactions occurring in the pattern. The AIN itself then describes the pattern as follows: It fixes which objects must be active for an interaction to be possible. By moving tokens it activates and deactivates the objects involved in the interactions and thus describes their effect on these objects. As usual, we graphically depict places as round nodes whereas transitions are depicted by rectangular nodes (cf. Fig. 8).

In our framework GUIs are generated by synthesizing an abstract description of a GUI from a repository of basic GUI building blocks. These blocks are given by GUI-fragments (GUIFs) and AINs. A GUIF is a single component that has a defined functionality and realizes a certain interaction. GUIFs describe reusable parts of a GUI, for example a drop-down menu. An AIN is a complex building block. It is a structural template which fixes the available objects but regards the transitions as placeholders. To obtain a description of a GUI each placeholder has to be substituted by a GUIF, that realizes the corresponding interaction directly, or (recursively) by an AIN[1], that describes a complex interaction pattern that realizes the interaction. The second case may arise, if on a more fine grained level of abstraction, for example, an interaction "search" has to be described by an AIN with two transitions which first requires the input of a search term followed by a selection from the listed search results. Since the building blocks may have to adhere to certain constraints each GUIF is linked to a usage context vector describing the contexts the GUIF is suitable for, i.e., the constraints that have to be satisfied if the GUIF is to be used.

[1]The substitution of a transition by an AIN has to obey certain rules.

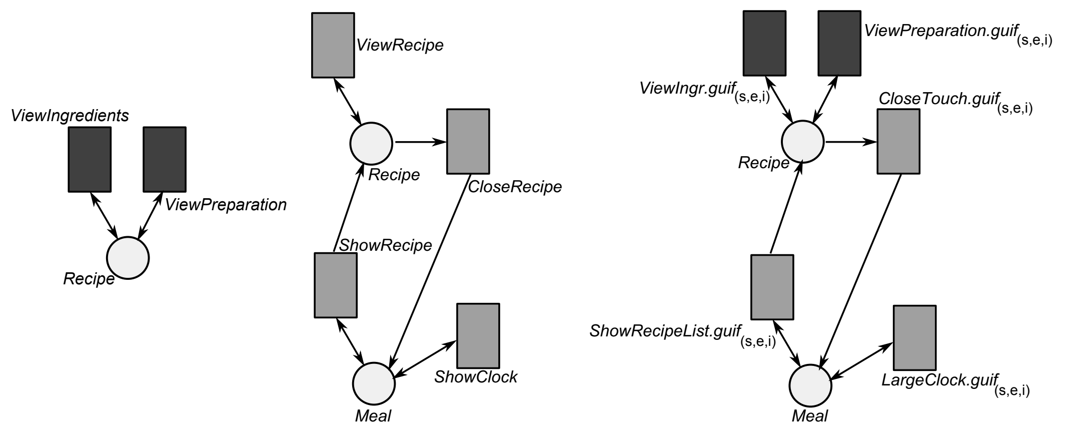
- **O** Meal
  - **I** Show Clock
    - **A** Show Clock$_{A1}$
      - **V** Show Clock$_{V1}$
        - **GUIF** $\mathit{LargeClock.guif}_{(s,e,i)}$
        - **GUIF** $\mathit{DesktopClock.guif}_{p}$
        - **AIN** $\mathit{DateTime.ain}$
  - **I** Show Recipe
    - **A** Show Recipe$_{A1}$
      - **V** Show Recipe$_{V1}$
        - **GUIF** $\mathit{ShowRecipeList.guif}_{(s,e,i)}$
- **O** Meal Plan
  - **I** View Meal
    - **A** View Meal$_{A1}$
      - **V** View Meal$_{V1}$
        - **AIN** $\mathit{ViewMeal.ain}$
  - **I** Edit Meal
    - **A** Edit Meal$_{A1}$
      - **V** Edit Meal$_{V1}$
        - **AIN** $\mathit{EditMeal.ain}$
- **O** Recipe
  - **I** View Recipe
    - **A** View Recipe Details
      - **V** View Ingr & Prep
        - **AIN** $\mathit{ViewIngr\&Prep.ain}$
  - **I** Close Recipe
    - **A** Close Recipe$_{A1}$
      - **V** Close Recipe$_{V1}$
        - **GUIF** $\mathit{CloseTouch.guif}_{(s,e,i)}$
        - **GUIF** $\mathit{CloseMouse.guif}_{p}$
  - **I** View Ingredients
    - **A** View Ingredients$_{A1}$
      - **V** View Ingredients$_{V1}$
        - **GUIF** $\mathit{ViewIngr.guif}_{(s,e,i)}$
        - **GUIF** $\mathit{IngrDetails.guif}_{p}$
  - **I** View Preparation
    - **A** View Preparation$_{A1}$
      - **V** View Preparation$_{V1}$
        - **GUIF** $\mathit{ViewPreparation.guif}_{(s,e,i)}$
        - **GUIF** $\mathit{AnimPreparation.guif}_{p}$

**Figure 7:** Excerpt of repository: Planning a meal

These context vectors can be use to synthesize a GUI-description which is optimized for certain given constraints. The repository has a hierarchical structure: The interactions used to label the transitions are divided into abstract interactions, alternatives, and variants. Each abstract interaction $i$ has a set of alternatives each of which realizes $i$. Alternatives can be used to differentiate between various methods to realize an abstract interaction, that have the same result but follow different approaches. For example, an abstract interaction for a search could have two alternatives: the first alternative describing a search by categories, the second alternative describing a search which uses a search mask. Each alternative $a$ has a set of variants each of which realizes $a$. Thus, the alternatives can be understood to logically group variants realizing the same abstract interaction. Each variant $v$ is directly realized by a set of GUIFs or by an AIN that describes a more complex interaction pattern realizing $v$ in the sense described above.

We consider a repository (Figure 7) from a medical scenario [12, 13] for synthesizing GUIs for web applications that support patients to keep diet after medical treatment, for example, by helping plan a meal. The repository's hierarchical structure is represented by layered lists containing the **O**bjects, abstract **I**nteractions, **A**lternatives, and **V**ariants. Figure 7 only depicts an excerpt of the repository. In particular, there are more than one alternative to every abstract interaction and more than one variant to every alternative which are not shown in the figure. Below the variants are listed the corresponding GUIFs or AINs. The indices of the GUIFs describe the usage contexts. A GUIF of usage context $(s,e,i)$, for example, is suitable for a **s**martphone used by **e**lderly people with **i**mpaired vision, whereas GUIFs of usage context $p$ are suitable for desktop **PC**s. The AINs $\mathit{ViewIngr\&Prep.ain}$ and $\mathit{ViewMeal.ain}$ are given in Figures 8(a), respectively 8(b).

The synthesis problem is now defined as follows: From a given AIN and a usage context vector we want to generate an abstract GUI-description that is optimized for the usage context vector by substituting each transition of the AIN by a suitable GUIF or AIN. Thus, given an AIN, we have to realize each of its transitions *separately* for the given usage context vector. The resulting adapted AIN where all its



(a) *ViewIngr&Prep.ain* (b) *ViewMeal.ain* (c) Resolved GUIF-AIN

**Figure 8:** Example AINs

transitions (in particular also those transitions that emerged due to substitution by further AINs) are realized by GUIFs is called a GUIF-AIN. The transitions of the AIN are realized by means of a recursive algorithm: If a transition labeled $i$ can directly be realized by a GUIF then it is substituted by the GUIF which must be executed whenever the transition is fired. Otherwise, if there is an AIN realizing $i$, then $i$ is substituted by the AIN, whose transitions then have to be recursively realized. Applying this procedure in order to realize *ViewMeal.ain*, results in the GUIF-AIN depicted in Figure 8(c). The transition labeled *ViewRecipe* of *ViewMeal.ain* first has to be replaced by *ViewIngr&Prep.ain* whose transitions can be directly realized by GUIFs.

In the following we explain how this approach to synthesizing GUIs can be mapped to inhabitation questions such that from the inhabitants a GUIF-AIN realizing the synthesis goal can be assembled: For each abstract interaction, alternative, and variant, as well as for each usage context, we introduce a fresh type constant. The hierarchical structure of abstract interactions, alternatives, and variants is represented by subtyping. We extend $\leq$ with the following additional conditions: We set $a \leq i$ for each abstract interaction $i$ and each of its corresponding alternatives $a$, and we set $v \leq a'$ for each alternative $a'$ and each of its corresponding variants $v$. We use intersections to represent the usage context vectors: Let $C$ be the set of all usage contexts[2]. A usage context given by the non-empty subset $\mathscr{C} \subseteq C$ is in principle represented by the intersection $\bigcap_{c\in\mathscr{C}} c$. However, to prevent name clashing we introduce the type constructor $\mathbf{uc}(\cdot)$ to encapsulate an intersection representing a usage context vector. We assume that $\mathbf{uc}$ is distributive with respect to intersection, i.e., $\mathbf{uc}(c \cap c') = \mathbf{uc}(c) \cap \mathbf{uc}(c')$. The repository of GUI building blocks described above is transformed into a type environment $\Gamma$ as follows. Each GUIF $x$ directly realizes the variant $v_x$ it is a child of. Therefore, $x$ must at least be given the type $v_x$. Because $x$ further has a usage context vector $\mathscr{C}_x$ we augment its type by $\mathbf{uc}(\bigcap_{c\in\mathscr{C}_x} c)$, i.e., we get $x : v_x \cap \mathbf{uc}(\bigcap_{c\in\mathscr{C}_x} c) \in \Gamma$. An AIN $f$ realizes the variant $v_f$ it is a child of if all transitions of $f$ are realized. Thus, if $f$ includes $n$ transitions labeled $i_1, \ldots, i_n$ then it must at least be given the type $i_1 \to \cdots \to i_n \to v_f$. Then, inhabiting $v_f$ using the combinator $f$ forces *all* arguments of $f$ also to be inhabited in accordance with the fact that the AIN $f$ is realized if all its transitions are realized. We still have to explain how usage context vectors are passed to the arguments of the function type representing the AIN. Consider a fixed context $\mathscr{C} \subseteq C$. The

[2]We assume that there are only finitely many usage contexts.

coding $f : i_1 \cap \mathbf{uc}(\bigcap_{c\in\mathscr{C}} c) \to \cdots \to i_n \cap \mathbf{uc}(\bigcap_{c\in\mathscr{C}} c) \to v_f \cap \mathbf{uc}(\bigcap_{c\in\mathscr{C}} c)$ passes the usage context $\mathscr{C}$ to all arguments. This coding ensures that in order to realize the variant $v_f$ supplied with the usage context vector $\mathscr{C}$ by using the combinator $f$ *all* $n$ arguments $i_1,\ldots,i_n$ must also be realized in a way which is optimized for $\mathscr{C}$. This reflects the fact that in order to construct a GUIF-AIN for a given usage context *all* its transitions must be realized according to this usage context. However, we of course do not want to restrict the combinators representing AINs to a single usage context vector. It must be possible to represent an AIN by a combinator that can pass an arbitrary usage context vector to its arguments because we do not know beforehand which usage context a GUI should be synthesized for. With monomorphic types (FCL) this would lead to the following coding:

$$f : (i_1 \to \cdots \to i_n \to v_f) \cap \bigcap_{c\in C} (\mathbf{uc}(c) \to \cdots \to \mathbf{uc}(c) \to \mathbf{uc}(c)) \in \Gamma$$

Using polymorphism ($\text{BCL}_0$) allows for a more succinct coding, because we may instantiate variables with an intersection representing exactly the usage context vector needed. Thus, we code the AIN $f$ by the combinator:

$$f : i_1 \cap \mathbf{uc}(\alpha) \to \cdots \to i_n \cap \mathbf{uc}(\alpha) \to v_f \cap \mathbf{uc}(\alpha) \in \Gamma$$

The synthesis goal consisting of an AIN $g$ with $m$ transitions labeled $k_1,\ldots,k_m$ and of a usage context vector $\mathscr{C}_g$ is represented by asking $m$ inhabitation questions $\Gamma \vdash ? : k_j \cap \mathbf{uc}(\bigcap_{c\in\mathscr{C}_g} c)$, for $1 \leq j \leq m$.

Part of the type environment $\Gamma_{OM}$ obtained by applying this translation to the example repository in Figure 7 is shown in Figure 9. As explained above, the subtype relation is extended for abstract interactions, alternatives, and variants. For example, the relations `ViewIngr&Prep` $\leq$ `ViewRecipeDetails` and `ViewRecipeDetails` $\leq$ `ViewRecipe` are derived from the repository. Recall that $C_{OM} = \{p,s,e,i\}$ is the set of usage contexts, which here contains usage contexts describing GUIFs that are suitable for desktop PCs, smartphones, elderly people, respectively people with impaired vision. For example, if a GUI to display the ingredients and the preparation instructions for a recipe should be realized for a smart-phone used by elderly users with impaired vision, the combinator `ViewIngr&Prep.ain` can be instantiated with the type:

$$\begin{aligned}&\texttt{ViewIngredients} \cap \mathbf{uc}(s \cap e \cap i) \to\\ &\quad \texttt{ViewPreparation} \cap \mathbf{uc}(s \cap e \cap i) \to \texttt{ViewIngr\&Prep} \cap \mathbf{uc}(s \cap e \cap i)\end{aligned}$$

In order to explain how to obtain a GUIF-AIN from an inhabitant consider an inhabitant $e$ of the type $j \cap \mathbf{uc}(\bigcap_{c\in\mathscr{C}_j} c)$ for an abstract interaction $j$. The corresponding GUIF or GUIF-AIN can recursively be constructed as follows: If $e = x$ where $x$ is a combinator representing the GUIF $x$ then a transition labeled $j$ is replaced by $x$. Otherwise, $e$ is of the form $f g_1 \ldots g_m$ where $f$ represents an AIN with $m$ transitions. In this case a transition labeled $j$ is replaced by the GUIF-AIN obtained from recursively replacing the transitions of $f$ by the GUIFs or GUIF-AINs corresponding to the terms $g_k$. To realize `ViewMeal.ain` for the context $\{s,e,i\}$, for example, we ask the four inhabitation questions:

$$\begin{aligned}&\Gamma_{OM} \vdash ? : \texttt{ShowRecipe} \cap \mathbf{uc}(s \cap e \cap i)\\ &\Gamma_{OM} \vdash ? : \texttt{ShowClock} \cap \mathbf{uc}(s \cap e \cap i)\\ &\Gamma_{OM} \vdash ? : \texttt{CloseRecipe} \cap \mathbf{uc}(s \cap e \cap i)\\ &\Gamma_{OM} \vdash ? : \texttt{ViewRecipe} \cap \mathbf{uc}(s \cap e \cap i)\end{aligned}$$

Figure 8(c) depicts the result.

The following section discusses (part of) a realization of this approach towards synthesizing GUIs.

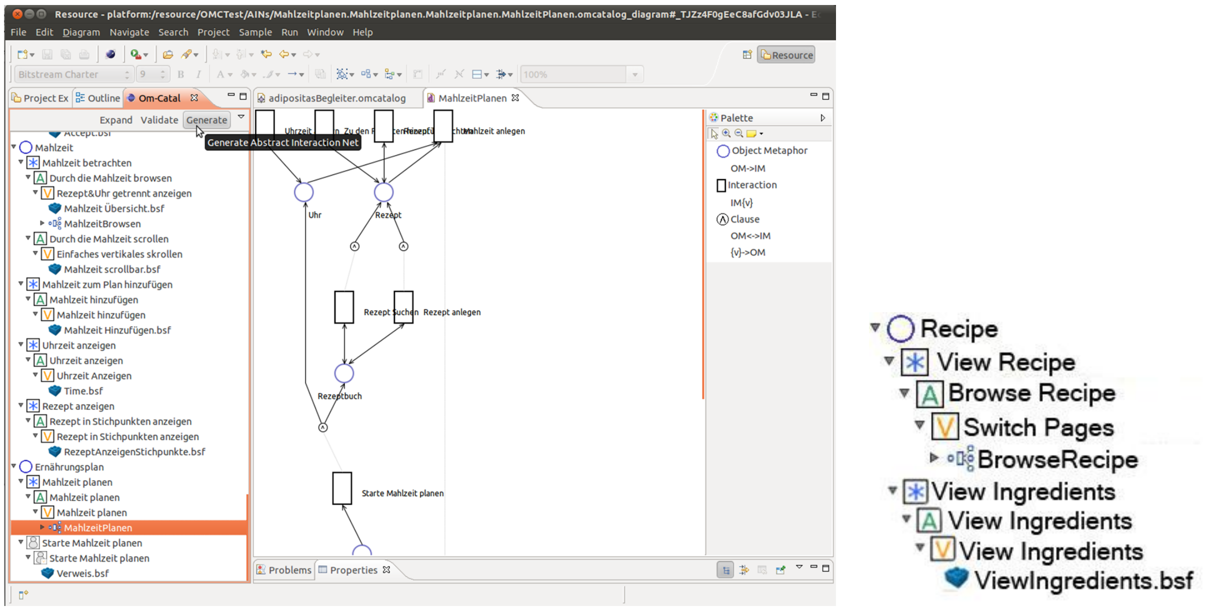
$$
\begin{array}{lll}
\Gamma_{OM} = \{ & \texttt{LargeClock.guif} & : \texttt{ShowClock}_{V1} \cap \mathbf{uc}(s \cap e \cap i), \\
 & \texttt{DesktopClock.guif} & : \texttt{ShowClock}_{V1} \cap \mathbf{uc}(p), \\
 & \texttt{ShowRecipeList.guif} & : \texttt{ShowRecipe}_{V1} \cap \mathbf{uc}(s \cap e \cap i), \\
 & \texttt{CloseTouch.guif} & : \texttt{CloseRecipe}_{V1} \cap \mathbf{uc}(s \cap e \cap i), \\
 & \texttt{CloseMouse.guif} & : \texttt{CloseRecipe}_{V1} \cap \mathbf{uc}(p), \\
 & \texttt{ViewIngr.guif} & : \texttt{ViewIngredients}_{V1} \cap \mathbf{uc}(s \cap e \cap i), \\
 & \texttt{IngrDetails.guif} & : \texttt{ViewIngredients}_{V1} \cap \mathbf{uc}(p), \\
 & \texttt{ViewPreparation.guif} & : \texttt{ViewPreparation}_{V1} \cap \mathbf{uc}(s \cap e \cap i), \\
 & \texttt{AnimPreparation.guif} & : \texttt{ViewPreparation}_{V1} \cap \mathbf{uc}(p), \\
 & \texttt{ViewMeal.ain} & : \texttt{ShowRecipe} \cap \mathbf{uc}(\alpha) \to \texttt{ShowClock} \cap \mathbf{uc}(\alpha) \to \\
 & & \quad \texttt{CloseRecipe} \cap \mathbf{uc}(\alpha) \to \texttt{ViewRecipe} \cap \mathbf{uc}(\alpha) \to \\
 & & \quad\quad \texttt{ViewMeal}_{V1} \cap \mathbf{uc}(\alpha), \\
 & \texttt{ViewIngr\&Prep.ain} & : \texttt{ViewIngredients} \cap \mathbf{uc}(\alpha) \to \\
 & & \quad \texttt{ViewPreparation} \cap \mathbf{uc}(\alpha) \to \\
 & & \quad\quad \texttt{ViewIngr\&Prep} \cap \mathbf{uc}(\alpha), \ldots \}
\end{array}
$$

**Figure 9:** Part of $\Gamma_{OM}$

## 5 Experiments

We presented a prototypical Prolog-implementation of the ATM shown in Figure 2 deciding inhabitation in BCL$_0$ [5]. It uses SWI-Prolog [25] and is based on a standard representation of alternation in logic programming [23]. This algorithm is used as the core search procedure in a synthesis-framework for GUIs that is based on the coding presented in the previous section. It consists of a Java implementation [10, 20] based on Eclipse [8] providing a suitable data-structure for the repository and a realization of the described translation of the repository into the type environment $\Gamma$. It offers a graphical user interface (Figure 10(a)) which allows for display and editing of the elements of the repository, posing synthesis-questions, and display and integration into the repository of the results. The constructed inhabitants are directly used to generate corresponding GUIF-AINs from the AINs and GUIFs in the repository. Figure 10(b) contains an enlarged extract of the repository displayed in Figure 10(a). It is a direct manifestation of the repository from which components are drawn for the synthesis of a GUI.

Using this implementation, we conducted some experiments on an extended version of the repository presented in the previous section. The repository contained 12 objects, 27 interactions, 31 alternatives, and 39 variants. There were 9 AINs and 35 GUIFs. In a first prototype the implementation did not treat usage contexts. Instead, a manual post-filtering procedure to identify the solutions best suited for the given usage context was used. Ignoring the usage contexts during inhabitation caused the number of solutions to be very large: Up to 20000 GUIF-AINs were found for some of the AINs in this rather small example. This shows that even for examples of a relatively small size the combinatory explosion may be immense underlining the need for an automation of composition in this case. Furthermore, the sheer number of 20000 solutions made the post-filtering cumbersome if not infeasible. In a second step we incorporated a pre-filtering of $\Gamma$, removing unneeded GUIFs. This resulted in a reduction of the number of suited solutions to approximately 500 which still proved infeasible regarding a composition of the components manually. Including usage contexts by means of intersection types and restricted polymorphism as described in the previous section further reduced the number to only a few solutions. The longest synthesis steps took two to three seconds.



(a) Repository with example AIN (b) Extract of repository

**Figure 10:** GUI for synthesis-framework

The presented implementation is only one part in a complete tool chain from design to generation for GUI-synthesis. Here we only focused on the synthesis of an abstract description of the GUI to be generated and its interaction processes. These processes are realized by specifying the necessary GUIFs. Then actual source-code for a web portal server is generated from the synthesized processes by wiring the GUIFs together in a predefined way, thus generating executable GUIs.

## 6 Related work

Our approach is related to adaptation synthesis via proof counting [11, 24], where semantic specifications at the type level are combined with proof search in a specialized proof system. The idea of adaptation synthesis [11] is closely related to our notion of composition synthesis. However our logic (bounded combinatory logic with intersection types) is different, and the algorithmic methods are different. In [11] the specification language used is a typed predicate logic, which is more expressive and more complex than the case of 0-bounded polymorphism ($\mathrm{BCL}_0$), being based on higher-order unification which is undecidable. The presence of intersection together with $k$-bounded polymorphism yields, of course, enormous theoretical expressive power (simulation of alternating space bounded Turing machines) and also complexity (nonelementary recursive when the bound $k$ is a parameter). A deeper comparison of the relative expressive power in practice must be left for future work. One example, though, which was brought up by one of our reviewers, is worth discussing here, since it points to a methodological point of some generality. The use of predicate logic in [11] allows, e.g., the specification of relations between the arguments of a function and its value. For instance, if $P$ is a ternary predicate on reals, we could have the specifications

$$\texttt{swapArgs} : (\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma \text{ and } \texttt{f} : \mathrm{R} \to \mathrm{R} \to \mathrm{R}$$

where `f` satisfies the property $\forall x,y:\mathtt{R}.\ P(x,y,fxy)$. If we ask for a composition $g$ satisfying the property $\forall x,y:\mathtt{R}.\ P(x,y,gyx)$, we get the result $g = \mathtt{swapArgs\ f}$. Such properties (as expressed by $P$) can foremostly be expressed in simple ways in our system when it is possible to name the properties and express their flow through the repository using intersections. In the case just shown, this can be easily done, by

$$\mathtt{swapArgs} : (\alpha \to \beta \to \gamma) \to ((\beta \to \alpha \to \gamma) \cap \mathit{swapped}) \text{ and } \mathtt{f} : \mathtt{R} \to \mathtt{R} \to \mathtt{R}$$

Asking for an inhabitant of $(\mathtt{R} \to \mathtt{R} \to \mathtt{R}) \cap \mathit{swapped}$ yields the solution shown. However, it may not always be easy to name complex relations in such a way that the semantics of them are fully captured. On the other hand, working with polymorphic types and type constructors in semantic intersection type components (as in, e.g., a semantic intersection type component $P(\alpha,\beta,\gamma)$) can be enormously expressive. Our lower bound construction in [7] gives theoretical evidence of this (the lower bound codes the tape of a space bounded Turing machine using such types). Further experience is needed to clarify this important class of questions.

Problems of synthesis from component libraries have been investigated within temporal logic and automata theory [14]. Our approach is fundamentally different, being based on type theory and combinatory logic, and a direct comparison is therefore precluded. However, the fact that both approaches lead to 2-EXPTIME complete problems (in our special case of 0-bounded polymorphism) might suggest that a more detailed comparison could be an interesting topic for further work.

## 7 Conclusion and further work

We have introduced the idea of composition synthesis based on combinatory logic with intersection types. Our work is ongoing, and we should emphasize that in the present paper we could only attempt to provide a first encounter with the ideas. There are many avenues for further work. Of foremost importance are optimization of the inhabitation algorithm and further experiments. Although the algorithm matches the worst-case lower bound, there are many interesting principles of optimization to be explored. For better scalability and functionality we have reimplemented the algorithm, using the Microsoft .NET-Framework (F# and C#) [2]. This implementation resulted in a tool called **(CL)S** [1, 17]. This allowed for a much greater flexibility than the Prolog-based implementation. Some highlights of this implementation are as follows: We parallelized the core procedure of the inhabitation algorithm, allowing for a simultaneous processing of inhabitation questions. This parallelization is an important step towards running our the inhabitation algorithm on multi-CPU architectures — for example, we deployed a version of this algorithm on a cluster with 1216 cores. Furthermore, we added various features improving usability: The input language is human-readable and closely oriented towards the formal type-language, and there are various graphical output formats, including a display of an execution graph illustrating the tasks processed during inhabitation and of the inhabitants produced. These graphical outputs proved very useful regarding debugging and analysis of the algorithms. We achieved first improvements with regard to optimization of the inhabitation algorithm as discussed above [6]. Finally, we added a mechanism which deals with cycles in the inhabitation procedure and thus, allows for a finite representation of infinite sets of inhabitants.

We further pursued the experimental application and evaluation of the ideas described here in a number of different areas: We used the inhabitation-based synthesis methodology presented here to control synthesis for Lego Mindstorms NXT robots, for example, we were able to synthesize simple pattern-follower programs for these robots with light- or ultrasonic-sensors. Currently, the methodology is applied in a factory planning project which we will report on, soon. Note that [17] contains a more theoretical exposition of the methodology presented here, also containing further examples. As mentioned above, a comparison to synthesis problems framed in temporal logics could be interesting.

**Acknowledgements**

The authors thank the referees for useful comments.

## References

[1] *Combinatory Logic Synthesizer — (CL)S*. `http://ls14-www.cs.tu-dortmund.de/index.php/CLS`.

[2] *.NET Development*. `http://msdn.microsoft.com/en-us/library/ff361664.aspx`.

[3] Ashok K. Chandra, Dexter C. Kozen & Larry J. Stockmeyer (1981): *Alternation*. *J. ACM* 28, pp. 114–133, doi:10.1145/322234.322243.

[4] Mario Coppo & Mariangiola Dezani-Ciancaglini. (1980): *An Extension of Basic Functionality Theory for Lambda-Calculus*. *Notre Dame Journal of Formal Logic* 21, pp. 685–693, doi:10.1305/ndjfl/1093883253.

[5] Boris Düdder, Moritz Martens & Jakob Rehof (2011): *Prototype Implementation of an Inhabitation Algorithm for FCL*$(\cap, \leq)$. Presentation at Types 2011 in Bergen, Norway.

[6] Boris Düdder, Moritz Martens & Jakob Rehof (2012): *Intersection Type Matching and Bounded Combinatory Logic (Extended Version)*. Technical Report 841, Faculty of Computer Science (TU Dortmund). `http://ls14-www.cs.tu-dortmund.de/index.php/Jakob_Rehof_Publications#Technical_Reports`.

[7] Boris Düdder, Moritz Martens, Jakob Rehof & Paweł Urzyczyn (2012): *Bounded Combinatory Logic*. In: *Computer Science Logic (CSL'12)*, *LIPIcs* 16, Leibniz-Zentrum fuer Informatik, pp. 243–258, doi:10.4230/LIPIcs.CSL.2012.243.

[8] Eclipse.org: *Eclipse Indigo (3.7) Documentation*. Available at `http://help.eclipse.org`.

[9] Tim Freeman & Frank Pfenning (1991): *Refinement Types for ML*. In: *ACM Conference on Programming Language Design and Implementation (PLDI)*, ACM, pp. 268–277, doi:10.1145/113445.113468.

[10] Oliver Garbe (2012): *Synthese von Benutzerschnittstellen mit einem Typinhabitationsalgorithmus*. Diploma thesis, Technical University of Dortmund.

[11] Christian Haack, Brian Howard, Allen Stoughton & Joe B. Wells (2002): *Fully Automatic Adaptation of Software Components Based on Semantic Specifications*. In: *AMAST'02*, *LNCS* 2422, Springer, pp. 83–98, doi:10.1007/3-540-45719-4_7.

[12] Thomas Königsmann (2011): *Compositional Modelling Ansatz zur Benutzerschnittstellengenerierung am Beispiel telemedizinischer Anwendungen*. Ph.D. thesis, Technical University of Dortmund.

[13] Thomas Königsmann & Reinholde Kriebel (2008): *Digitale Gesundheitsbegleiter am Beispiel der Adipositas-Nachsorge*. In: *Proceedings of AAL 2008*, Verband der Elektrotechnik, Elektronik, Informationstechnik, VDE-Verlag.

[14] Y. Lustig & M. Y. Vardi (2009): *Synthesis from Component Libraries*. In: *FOSSACS*, *LNCS* 5504, Springer, pp. 395–409, doi:10.1007/978-3-642-00596-1_28.

[15] Dale Miller, Gopalan Nadathur, Frank Pfenning & Andre Scedrov (1991): *Uniform Proofs as a Foundation for Logic Programming*. *Ann. Pure Appl. Logic* 51(1-2), pp. 125–157, doi:10.1016/0168-0072(91)90068-W.

[16] Garrel Pottinger (1980): *A Type Assignment for the Strongly Normalizable Lambda-Terms*. In J. Hindley & J. Seldin, editors: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, pp. 561–577.

[17] Jakob Rehof (2013): *Towards Combinatory Logic Synthesis*. In: *Proceedings of BEAT'13*, ACM.

[18] Jakob Rehof & Paweł Urzyczyn (2011): *Finite Combinatory Logic with Intersection Types*. In: *TLCA*, *Lecture Notes in Computer Science* 6690, Springer, pp. 169–183, doi:10.1007/978-3-642-21691-6_15.

[19] Jakob Rehof & Paweł Urzyczyn (2012): *The Complexity of Inhabitation with Explicit Intersection*. In Robert L. Constable & A. Silva, editors: *Kozen Festschrift*, LNCS 7230, pp. 256–270, doi:10.1007/978-3-642-29485-3_16.

[20] Eugen Reinke (2011): *Konzeption und Entwicklung eines Tools zur Modellierung von User-Interface-Spezifikationsbausteinen im Rahmen des "Compositional Modeling"-Ansatzes*. Diploma thesis, Technical University of Dortmund.

[21] Sylvain Salvati (2009): *Recognizability in the Simply Typed Lambda-Calculus*. In H. Ono, M. Kanazawa & R. J. G. B. de Queiroz, editors: *WoLLIC*, *LNCS* 5514, Springer, pp. 48–60, doi:10.1007/978-3-642-02261-6_5.

[22] Sylvain Salvati, Giulio Manzonetto, Mai Gehrke & Henk Barendregt (2012): *Loader and Urzyczyn Are Logically Related*. In: *ICALP 12, Automata, Languages, and Programming - 39th International Colloquium, Warwick, UK*, *LNCS* 7392, Springer, pp. 364–376, doi:10.1007/978-3-642-31585-5_34.

[23] Ehud Y. Shapiro (1984): *Alternation and the Computational Complexity of Logic Programs*. *J. Log. Program.* 1(1), pp. 19–33, doi:10.1016/0743-1066(84)90021-9.

[24] J. B. Wells & Boris Yakobowski (2005): *Graph-Based Proof Counting and Enumeration with Applications for Program Fragment Synthesis*. In Sandro Etalle, editor: *LOPSTR 2004*, *LNCS* 3573, Springer, pp. 262–277, doi:10.1007/11506676_17.

[25] Jan Wielemaker, Tom Schrijvers, Markus Triska & Torbjörn Lager (2010): *SWI-Prolog*. *Computing Research Repository* abs/1011.5332.