Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI 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 which is based on combinatory logic with intersection types. The idea is that components are modelled as typed combinators, and an algorithm for inhabitation — is there a combinatory term e with type τ relative to an environment Γ? — can be used to synthesize compositions. Here, Γ represents the repository in the form of typed combinators, τ specifies the synthesis goal, and e is the synthesized program. We illustrate our approach by 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 [12, 13, 4] we have laid the theoretical foundations for understanding algorithmics and complexity of decidable inhabitation in subsystems of the intersection type system [2]. 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 Γ given as part of the input. This problem is undecidable for combinatory logic, even in simple types, see [4]. We have introduced finite and bounded combinatory logic with intersection types in [12, 4] as a possible foundation for type-based composition synthesis. *Finite combinatory logic* (abbreviated FCL) [12] arises from combinatory logic by restricting combinator types to be monomorphic, and *k-bounded combinatory logic* (abbreviated BCL_k) [4] 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 [12], 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

Submitted to:
ITRS 2012

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 Γ of typed combinators and a type τ, does there exist a combinatory term e such that $\Gamma \vdash e : \tau$? For applications in synthesis, we consider Γ as a repository of components represented only by their names (combinators) and their types (intersection types), and τ is seen as the specification of a synthesis goal. An inhabitant e is a program obtained by applicative combination of components in Γ. The inhabitant e is automatically constructed (synthesized) by the inhabitation algorithm. For applications to synthesis, where the repository Γ 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* [12, 4]. 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 term can be uniquely written as $xe_1 \ldots e_n$ for some $n \geq 0$. Sometimes we may also write $x(e_1, \ldots, e_n)$ instead of $xe_1 \ldots e_n$. Types, ranged over by τ, σ, etc. are defined by

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

where $a, b, c, \ldots$ range over atoms comprising *type constants* from a finite set $\mathbb{A}$, a special constant ω, and *type variables* from a disjoint denumerable set $\mathbb{V}$ ranged over by $\alpha, \beta, \gamma, \ldots$ 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 [11, 2] and is said to have τ and σ as components. We sometimes write $\bigcap_{i=1}^{n} \tau_i$ for an intersection with $n \geq 1$ components. If $\tau = \tau_1 \to \ldots \to \tau_n \to \sigma$ we write $\sigma = tgt_n(\tau)$ and $\tau_i = arg_i(\tau)$ for $i \leq n$ and we say that σ is a target type of τ and τ_i are argument types of τ. A type of the form $\tau = \tau_1 \to \ldots \to \tau_n \to a$ 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 τ there is an equivalent organized type $\bar{\tau}$ that is computable in polynomial time [13]. 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 σ and by $\| \sigma \|$ the path length of σ which is defined to be the maximal length of a path in σ. Define the set $\mathbb{T}_0$ of *level-0 types* by $\mathbb{T}_0 = \{\bigcap_{i \in I} a_i | a_i \text{ an atom}\}$. 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 Γ and τ (always including the constant ω). 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 Γ is a finite set of type assumptions of the form $x : \tau$. Intersection types come with a natural notion of subtyping $\leq$ as defined in [2] and used in the systems of [12, 4]. Subtyping includes the axiom $\tau_1 \cap \tau_2 \leq \tau_i$ and therefore contains the intersection elimination rule. The subtyping relation is decidable in polynomial time [12].

The type rules for 0-*bounded combinatory logic with intersection types and subtyping*, denoted $\text{BCL}_0(\cap, \leq)$, as presented in [4], 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, BCL_0 allows a limited form of polymorphism of combinators in Γ, where type variables can be instantiated with atomic types or intersections of such. *Finite combinatory logic with intersection types and subtyping*, denoted $\mathrm{FCL}(\cap,\leq)$, as presented in [12], is the monomorphic restriction where the substitutions S in rule (var) of Figure 1 are required to be the identity and hence simplifies to the axiom $\Gamma, x:\tau \vdash x:\tau$.

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

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

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

We consider the inhabitation problem: *Given an environment Γ and a type τ, 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 [12] it is shown that deciding inhabitation in $\mathrm{FCL}(\cap,\leq)$ 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) [1]. In [4] 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 $\mathrm{BCL}_0(\cap,\leq)$ is 2-EXPTIME-complete. The lower bound for 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 ($\mathrm{FORALL}(i=1\ldots n)\,s_i$). 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 $\mathrm{FORALL}(i=1\ldots n)\,s_i$ 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 [4] for further details.

3 Synthesis from a Component Repository

In this section we briefly summarize some main points of our methodology for composition synthesis.

Semantic specification. It is well known that intersection types can be used to specify deep semantic properties. The system characterizes the strongly normalizing terms [11, 2], the inhabitation problem is closely related to the λ-definability problem [15], and our work on bounded combinatory logic [12, 4] 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 Γ_e and a type τ_e such that e is the unique term with $\Gamma_e\vdash e:\tau_e$ (see [12]).

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

Input : Γ, τ

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 $\text{BCL}_0(\cap,\leq)$

with code, because 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

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

where native types ($real$, $real \times real, \ldots$) are qualified, using intersections with semantic types (in the example, $\mathbf{Cart}, \mathbf{Pol}, \mathbf{Iso}$) expressing (relative to a given conceptual taxonomy) interesting domain-specific properties of the function (combinator) 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., $\mathbf{Cart}, \mathbf{Pol}$ are features of coordinates, $\mathbf{Iso}$ is a feature of functions) whose elements can be mapped onto the native API using intersections, at any level of the type structure. We feel that we are only beginning to realize the potential of this point of view.

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 Γ, and we want to synthesize compositions of such functions represented as types (in some of our applications they could, for example, be refinement types [6]). 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. In BCL_0, we can represent g with 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 Γ. Through level-0 polymorphism, the action of g is thereby fully specified. Generally, the level BCL_0 is already very expressive, because we can atomize type structure by introducing type names for complex types, if needed.

4 Examples

In this section we focus on the application of inhabitation in bounded combinatory logic. 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. In this section 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 behaviour. 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 3 shows a type environment Γ which models a GUI programming scenario for an abstract windowing system. Further, we define the subtyping relations $layoutDesktop \leq layoutObj$ and $layoutPDA \leq layoutObj$. 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** written in bold font express the current state of the protocol. Type inhabitation can now be used to synthesize the program described above by asking the inhabitation question $\Gamma \vdash ? : \mathbf{closed}$. The inhabitants

$$e_1 := closeWindow(interact(createControls(openWindow(init), layoutDesktopPC)))$$

$$e_2 := closeWindow(interact(createControls(openWindow(init), layoutPDAPhone)))$$

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 behaviour. These terms are type correct and in addition semantically correct (cf. Wells et al. [8]), because all specification axioms defined by the semantic types are satisfied. Generally, our lower-bound techniques [12, 4] show how we can express very complicated protocols inside $\mathrm{FCL}(\cap, \leq)$ (alternating tree automata) and $\mathrm{BCL}_0(\cap, \leq)$ (exponential space bounded ATMs).

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

Figure 3: Type environment Γ 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 [9], thereby enabling automatic synthesis of GUI-applications from a repository of

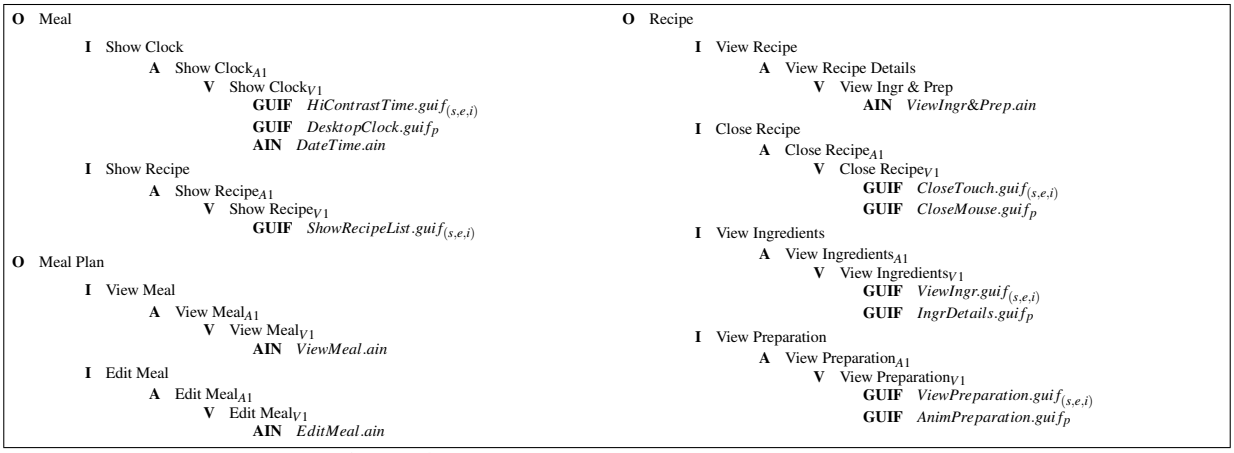
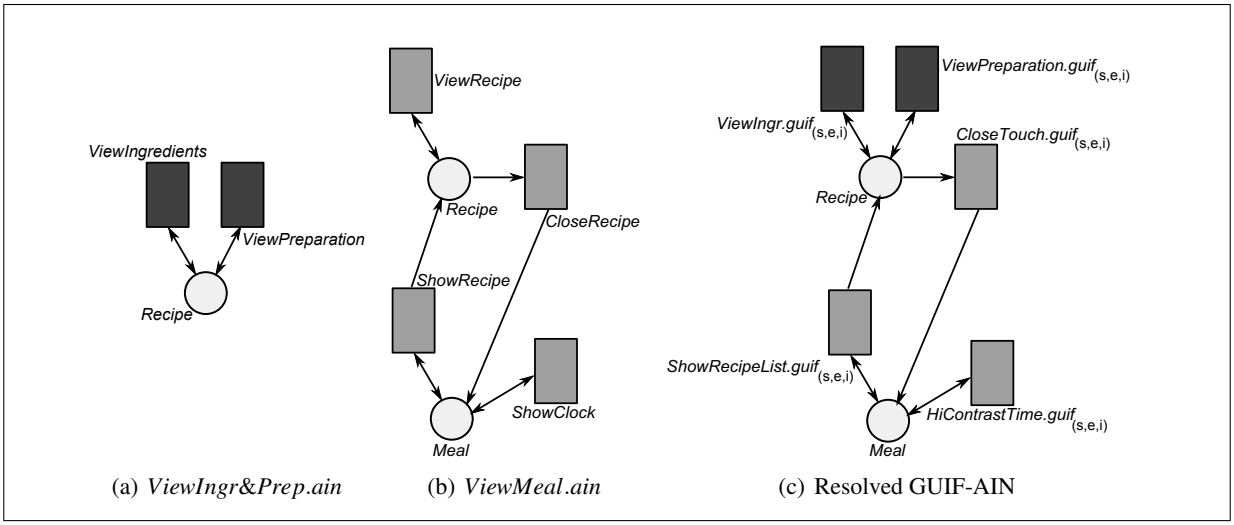


Figure 4: Excerpt of repository: Planning a meal

components. The main point we wish to illustrate is the integration of inhabitation in a complex software synthesis framework. In this framework, GUIs are generated by synthesizing an abstract description of a GUI, which is optimized for given constraints, from a repository of basic GUI building blocks. These blocks are given by GUI-fragments (GUIFs), each of which is a single component realizing a certain defined functionality. They describe reusable parts of a GUI, for example a drop-down menu. Each GUIF is linked to a usage context vector describing the contexts the GUIF is suitable for. The repository defines the objects and interactions that may be available in a GUI. The repository has a hierarchical structure: The interactions are divided into abstract interactions, alternatives, and variants. Each abstract interaction i has a set of alternatives each of which realizes i. Each alternative a has a set of variants each of which realizes a. Each variant v has a set of GUIFs and a set of abstract interaction nets (AINs). A GUIF directly realizes v. An AIN is an extended Petri net, where places and transitions are labelled with objects, respectively, interactions. Such an AIN can be understood as a structural template which fixes the available objects (i.e., the labels of the places in the AIN), whereas the transitions are placeholders that have to be substituted by further GUIFs or (recursively) AINs realizing the interaction identified by the label of the transition. An AIN describes an interaction process that realizes the corresponding variant v if all its transitions are fully realized.

We consider a repository (Figure 4) from a medical scenario [9, 10] 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 4 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. 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 **P**Cs. The AINs *ViewIngr&Prep.ain* and *ViewMeal.ain* are given in Figures 5(a), respectively 5(b).

The synthesis problem is defined as follows: From a given AIN and a usage context vector we want to generate an adapted AIN optimized for the usage contexts where each transition is realized by a GUIF. Thus, given an AIN, we have to realize each of its transitions *separately* for the given usage context vector. The resulting adapted AIN is called a GUIF-AIN. The transitions of the AIN are realized by means of a recursive algorithm: If a transition labelled i can directly be realized by a GUIF then it is



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

Figure 5: Example AINs

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[1] 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 5(c). The transition labelled *ViewRecipe* of *ViewMeal.ain* first has to be replaced by *ViewIngr&Prep.ain* whose transitions can be directly realized by GUIFs.

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. The repository is represented by a type environment Γ and the synthesis goal by the target type for the inhabitation. For each abstract interaction, alternative, and variant as well as for each usage context we introduce a fresh type variable. Formally, let C be a finite set containing all usage contexts. A non-empty subset $\mathscr{C} \subseteq C$ is a usage context vector. The hierarchical structure of abstract interactions, alternatives, and variants is represented by subtyping. We extend $\leq$ with the following additional conditions: $a \leq i$ for each abstract interaction i and each of its corresponding alternatives a and $v \leq a'$ for each alternative a' and each of its corresponding variants v. The synthesis goal consisting of an AIN g with m transitions labelled $k_1, \ldots, k_m$ and of a usage context vector $\mathscr{C}_g$ is represented by asking m inhabitation questions $\Gamma \vdash ? : k_j \cap \bigcap_{c \in \mathscr{C}_g} c$ for $1 \leq j \leq m$. Each GUIF x directly realizes the variant v_x it is a child of. Therefore, x is represented by the combinator $x : v_x$. Because every GUIF further has a usage context vector $\mathscr{C}_x$ we augment the type of the combinator x by an intersection of the usage contexts in $\mathscr{C}_x$. We get $x : v_x \cap \bigcap_{c \in \mathscr{C}_x} c \in \Gamma$. An AIN f realizes the variant v_f it is a child of if all transitions of f are realized. If f includes n transitions labelled $i_1, \ldots, i_n$ then it is represented by the combinator $f : i_1 \to \ldots \to i_n \to v_f$. Thus, 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. From this coding, however, the question arises how the usage context vector $\mathscr{C}_g$, given by the synthesis goal, is passed to the arguments of a function type. Let $f : i_1 \to \ldots \to i_n \to v_f$ model an AIN in the repository. The coding $f : i_1 \cap \bigcap_{c \in \mathscr{C}_g} c \to \ldots \to i_n \cap \bigcap_{c \in \mathscr{C}_g} c \to v_f \cap \bigcap_{c \in \mathscr{C}_g} c$ passes $\mathscr{C}_g$ to all arguments. This coding ensures that in order to inhabit the target type v_f supplied with the usage context vector $\bigcap_{c \in \mathscr{C}_g} c$ using the combinator f *all* n arguments $i_1, \ldots, i_n$ must also be inhabited in a way which is optimized for the same usage context vector. This reflects the fact that in order to

[1] This substitution has to obey certain rules.

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

Figure 6: Part of Γ_{OM}

construct a GUIF-AIN for a given usage context *all* its transitions must be realized according to this usage context. Since we do not want to restrict the possible usage contexts of an AIN we must provide every combinator in Γ that represents an AIN with every combination of possible usage context vectors. With monomorphic types (FCL) this would lead to the following coding:

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

However, for every concrete inhabitation question, in which f occurs, only one context vector is needed. Using polymorphism (BCL_0) allows for a simple coding, because we may instantiate variables with intersections representing usage context vectors. Now, we code f by the combinator

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

where **uc** is a type constructor for (the appropriate kind of) usage contexts. In order to inhabit variant v_f provided with a certain usage context vector (e.g., the variant should be realized for a smart-phone used by elderly users with impaired vision), rule (var) now allows to instantiate variable α with the intersection of the needed contexts (i.e., with $s \cap e \cap i$).

Part of the type environment Γ_{OM} obtained by applying this translation to the example repository in Figure 4 is shown in Figure 6. Note that $C_{OM} = \{p, s, e, i\}$. Using rule (var) the combinator *ViewIngr&Prep.ain* can be given the type

$$\mathit{ViewIngredients} \cap \mathbf{uc}(s \cap e \cap i) \to \mathit{ViewPreparation} \cap \mathbf{uc}(s \cap e \cap i) \to \mathit{ViewIngr\&Prep} \cap \mathbf{uc}(s \cap e \cap i),$$

for example. The subtype relation is extended for abstract interactions, alternatives, and variants, as described. For example, the relations $\mathit{ViewIngr\&Prep} \leq \mathit{ViewRecipeDetails}$ and $\mathit{ViewRecipeDetails} \leq \mathit{ViewRecipe}$ are derived from the repository.

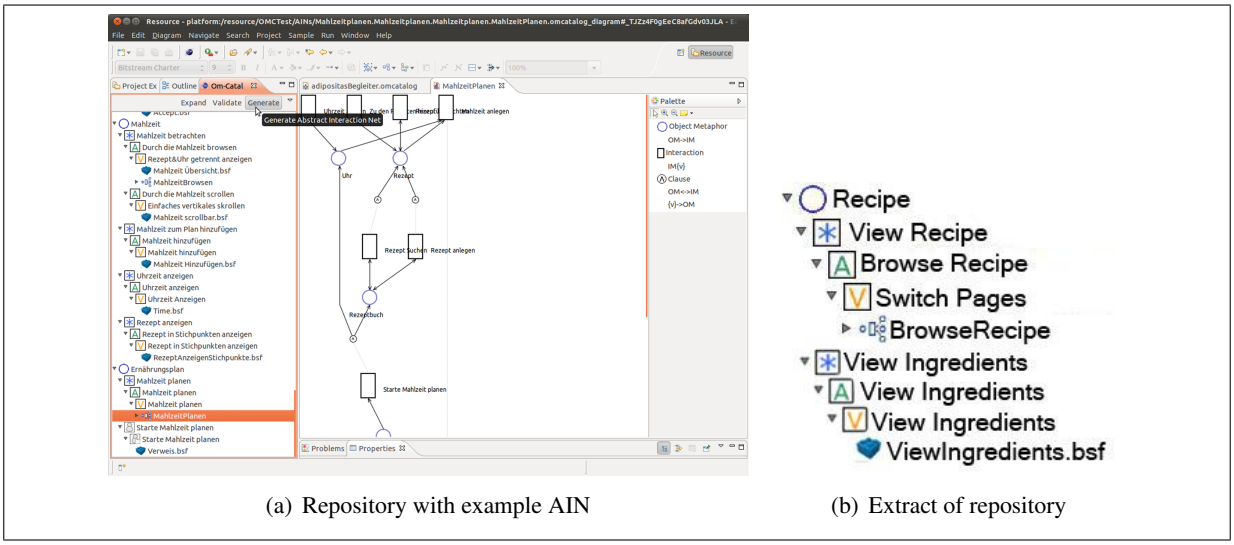
In order to explain how to obtain a GUIF-AIN from an inhabitant consider an inhabitant e of $j \cap \bigcap_{c \in \mathscr{C}_j} c$ for an abstract interaction j. The corresponding GUIF or GUIF-AIN can recursively be constructed as follows: If $e = x$ where x is a combinator representing the GUIF x then a transition labelled j

is replaced by x. Otherwise, e is of the form $fg_1 \dots g_m$ where f represents an AIN with m transitions. In this case a transition labelled 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 $\Gamma \vdash ? : ShowRecipe \cap \mathbf{uc}(s \cap e \cap i)$, $\Gamma \vdash ? : ShowClock \cap \mathbf{uc}(s \cap e \cap i)$, $\Gamma \vdash ? : CloseRecipe \cap \mathbf{uc}(s \cap e \cap i)$, and $\Gamma \vdash ? : ViewRecipe \cap \mathbf{uc}(s \cap e \cap i)$. Figure 5(c) depicts the result.

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

5 Implementation

We presented a prototypical Prolog-implementation of the ATM shown in Figure 2 deciding inhabitation in BCL_0 [3]. It uses SWI-Prolog [17] and is based on a standard representation of alternation in logic programming [16]. 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 [7, 14] based on Eclipse [5] providing a suitable data-structure for the repository and a realization of the described translation of the repository into the type environment Γ. It offers a graphical user interface (Figure 7(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 7(b) contains an enlarged extract of the repository displayed in Figure 7(a). It is a direct manifestation of the repository from which components are drawn for the synthesis of a GUI.



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

Figure 7: GUI for synthesis-framework

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 solutions were found in some cases for this rather small example), making the post-filtering cumbersome if not infeasible. In a second step we incorporated a pre-filtering of Γ removing unneeded GUIFs. This resulted in a reduction of the number of suited solutions to approximately 500. 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 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.

References

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

[2] 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.

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

[4] Boris Düdder, Moritz Martens, Jakob Rehof & Paweł Urzyczyn (2012): *Bounded Combinatory Logic (Extended Version)*. Technical Report 840, Technical University of Dortmund. A paper based on this report has been submitted for publication.

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

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

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

[8] 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.

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

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

[11] 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.

[12] Jakob Rehof & Paweł Urzyczyn (2011): *Finite Combinatory Logic with Intersection Types*. In: *TLCA*, *Lecture Notes in Computer Science* 6690, Springer, pp. 169–183.

[13] 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.

[14] 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.

[15] 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.

[16] Ehud Y. Shapiro (1984): *Alternation and the Computational Complexity of Logic Programs*. *J. Log. Program.* 1(1), pp. 19–33.

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