technische universität
dortmund

department of
computer science

TECHNICAL REPORTS IN COMPUTER SCIENCE

Technical University of Dortmund

Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Composition Synthesis (Extended Version)

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

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

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

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

(Partly supported by grant N N206 355836 from the Ministry of Science and Higher Education)

Number: 842
October 2012

ABSTRACT

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 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 examples, including an application to synthesis from GUI-components.

CONTENTS

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 [15, 16, 5] we have laid the theoretical foundations for understanding algorithmics and complexity of decidable inhabitation in subsystems of the intersection type system [3]. 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 [5]. We have introduced finite and bounded combinatory logic with intersection types in [15, 5] as a possible foundation for type-based composition synthesis. *Finite combinatory logic* (abbreviated FCL) [15] arises from combinatory logic by restricting combinator types to be monomorphic, and *k-bounded combinatory logic* (abbreviated BCL_k) [5] 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 [15], 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 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* [15, 5]. We consider applicative terms ranged over by e, etc. and defined as

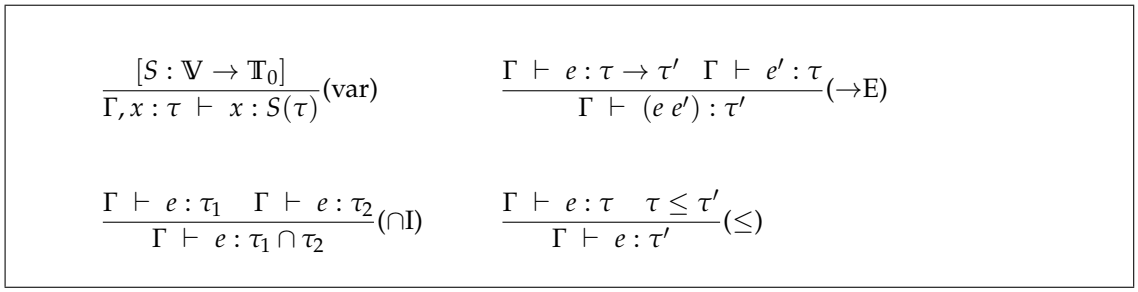
$$e ::= x \mid (e\ e'),$$

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 $xe_1 \dots e_n$ 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 τ, σ, 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 $\mathbb{A}$, a special constant ω, and *type variables* from a disjoint denumerable set $\mathbb{V}$ ranged over by $\alpha, \beta, \gamma, \dots$ 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 [14, 3] 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 $\tau = \tau_1 \to \dots \to \tau_n \to \sigma$ 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 $\tau = \tau_1 \to \dots \to \tau_n \to a$ 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 [16]. 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 $\parallel \sigma \parallel$ 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 [3] and used in the systems of [15, 5]. 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 [15].

The type rules for 0-*bounded combinatory logic with intersection types and subtyping*, denoted $\text{BCL}_0(\cap, \leq)$, as presented in [5], are given in Figure 2.1. 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



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

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

Figure 2.1: $\text{BCL}_0(\cap, \leq)$

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 $\text{FCL}(\cap, \leq)$, as presented in [15], is the monomorphic restriction where the substitutions S in rule (var) of Figure 2.1 are required to be the identity and hence simplifies to the axiom $\Gamma, x : \tau \ \vdash \ x : \tau$.

We consider the relativized 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 [15] 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) [2]. In [5] 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 BCL_0 is by reduction from acceptance of an exponential space bounded ATM.

The 2-EXPTIME (alternating exponential space) algorithm is shown in Figure 2.2. In Figure 2.2 we use shorthand notation for ATM-instruction sequences starting from existential states (CHOOSE ...) and instruction sequences starting from universal states ($\text{FORALL}(i = 1 \ldots n)\, s_i$). 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 $\text{FORALL}(i = 1 \ldots n)\, s_i$ 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 [5] for further details.

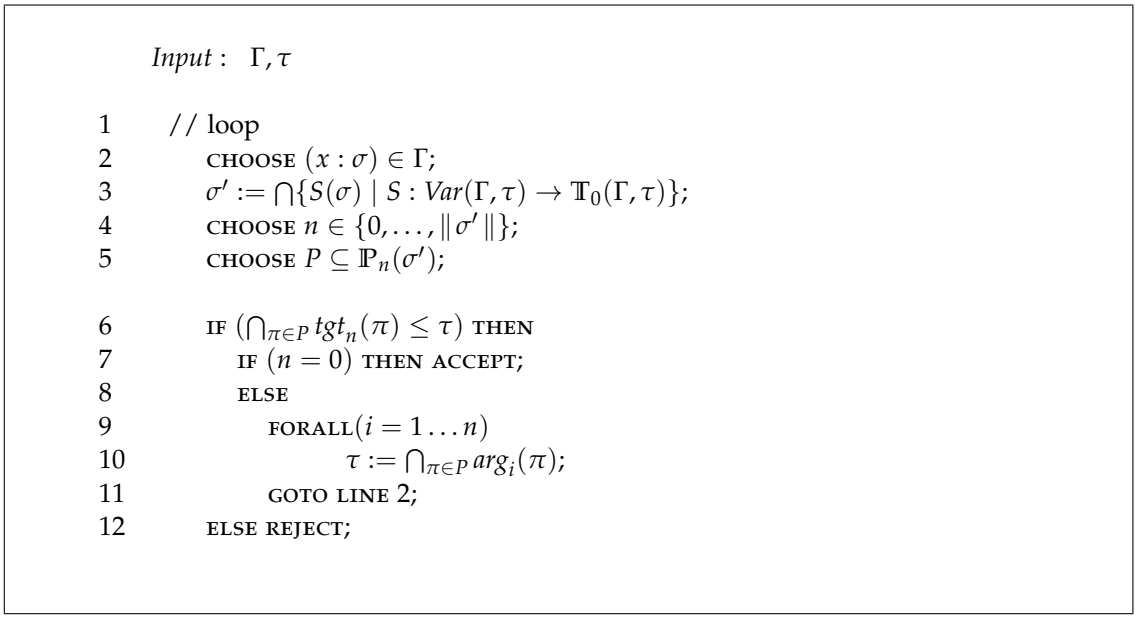


Figure 2.2: Alternating Turing machine $\mathcal{M}$ deciding inhabitation for $\text{BCL}_0(\cap, \leq)$

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.

3.1 BASIC PRINCIPLES

Semantic specification

It is well known that intersection types can be used to specify deep semantic properties in the λ-calculus. The system characterizes the strongly normalizing terms [14, 3], the inhabitation problem is closely related to the λ-definability problem [18, 19], and our work on bounded combinatory logic [15, 5] 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 [15]).

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

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

where native types (`real`, `real` $\times$ `real`, ...) are qualified, using intersections with semantic types (in the example, *Cart*, *Pol*, *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., *Cart*, *Pol* are features of coordinates, *Iso* is a feature of functions) whose elements can be mapped onto the native API using intersections, at any level of the type structure.

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 [7]). 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, as we can code arbitrary exponential space bounded alternating Turing machines at that level [5].

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 Γ 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. [13]. Under this viewpoint, solving the inhabitation problem $\Gamma \vdash ? : \tau$ means evaluating the program Γ against the goal τ: each typed combinator $F : \sigma$ in Γ names a single logical "rule" (type σ) in an implicational logic, and the repository Γ (a collection of such rules) constitutes a logic "program", which, when given a goal formula τ (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, Γ). Indeed, we can view the search procedure of the inhabitation algorithm shown in Figure 2.2 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 3.1, 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 $\mathtt{D}((\mathtt{R},\mathtt{R}),\mathtt{R},\mathtt{R})$ 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.

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

Figure 3.1: Section of repository implementing a tracking service (native API)

Now, the problem with the standard, native API shown in Figure 3.1 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 3.2 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.

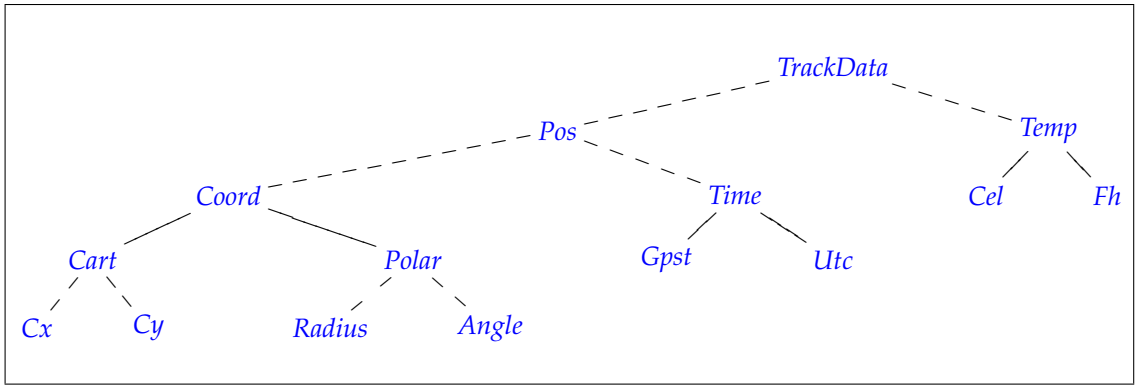


Figure 3.2: Semantic structures

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

Figure 3.3: Repository with semantic specifications

In Figure 3.3 we show the repository of Figure 3.1 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 (α, β, γ) 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 (α standing for

the semantic qualification of the coordinate component, β 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 [7] 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 3.3 considered as a combinatory type environment Γ 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

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

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 4.1 shows a type environment Γ 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. Wells et al. [9]), because all specification axioms defined by the semantic types are satisfied. Generally, our lower-bound techniques [15, 5] show how we can express very complicated protocols inside $\textsc{fcl}(\cap, \leq)$ (alternating tree automata) and $\textsc{bcl}_0(\cap, \leq)$ (exponential space bounded ATMs).

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

Figure 4.1: 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 [10], 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.

In our 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 labeled 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 4.2) from a medical scenario [10, 11] 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 4.2 only depicts

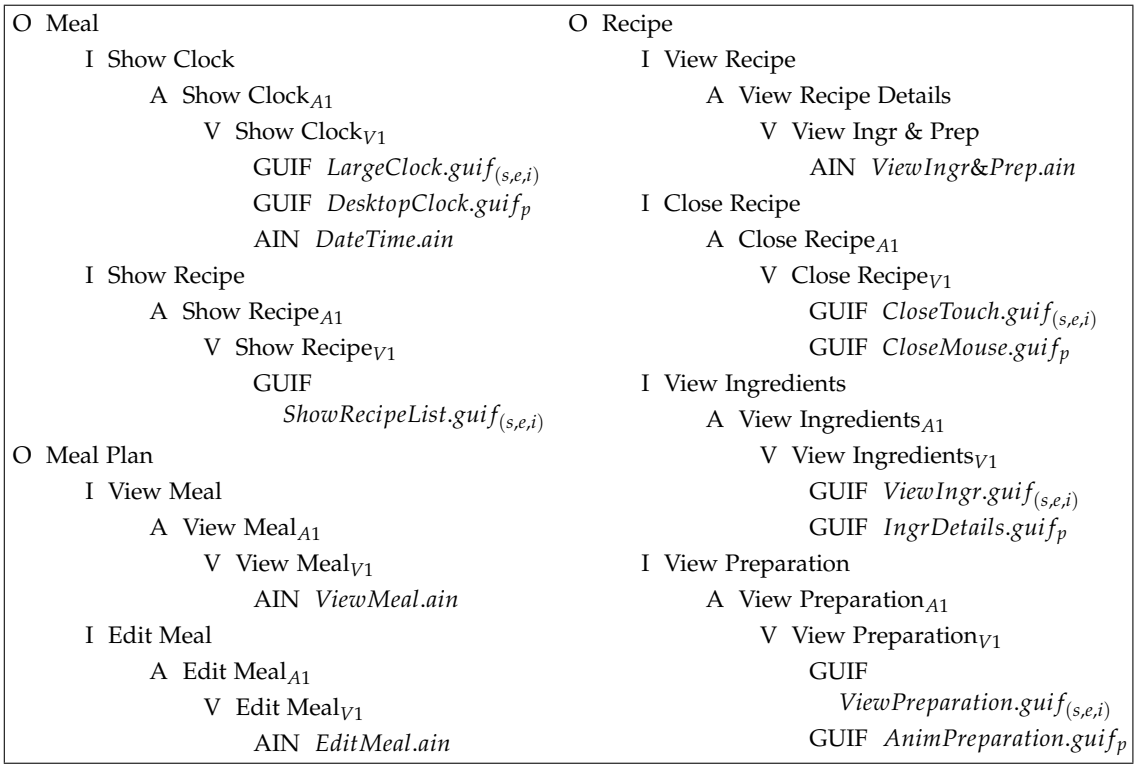
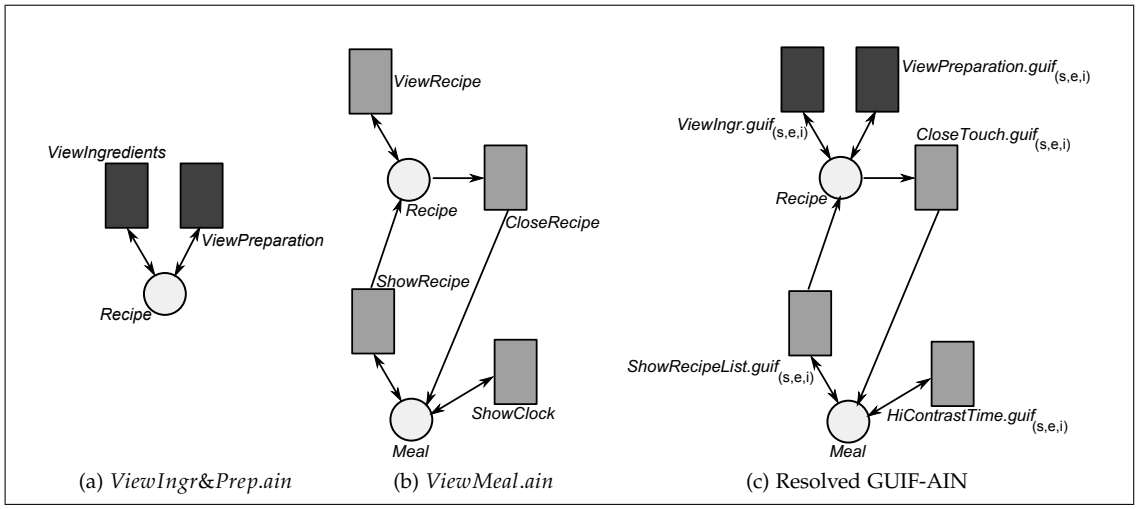


Figure 4.2: Excerpt of repository: Planning a meal

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 4.3a, respectively 4.3b.

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 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[1] by the AIN, whose transitions then have to be recursively realized. Applying this procedure

1 This substitution has to obey certain rules.



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

Figure 4.3: Example AINs

in order to realize *ViewMeal.ain*, results in the GUIF-AIN depicted in Figure 4.3c. The transition labeled *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 $\mathcal{C} \subseteq C$ 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 labeled $k_1, \ldots, k_m$ and of a usage context vector $\mathcal{C}_g$ is represented by asking m inhabitation questions $\Gamma \vdash ? : k_j \cap \bigcap_{c \in \mathcal{C}_g} c$, 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 $\mathcal{C}_x$ we augment the type of the combinator x by an intersection of the usage contexts in $\mathcal{C}_x$. We get $x : v_x \cap \bigcap_{c \in \mathcal{C}_x} c \in \Gamma$. 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 labeled $i_1, \ldots, i_n$ 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 $\mathcal{C}_g$, 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 $f : i_1 \cap \bigcap_{c \in \mathcal{C}_g} c \to \ldots \to i_n \cap \bigcap_{c \in \mathcal{C}_g} c \to v_f \cap \bigcap_{c \in \mathcal{C}_g} c$ passes $\mathcal{C}_g$ to all arguments. This coding ensures that in order to inhabit the target type v_f supplied with the usage context vector $\bigcap_{c \in \mathcal{C}_g} c$ 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 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 4.2 is shown in Figure 4.4. 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, using rule (var) the combinator `ViewIngr&Prep.ain` can be given the type

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

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

In order to explain how to obtain a GUIF-AIN from an inhabitant consider an inhabitant e of $j \cap \bigcap_{c \in \mathcal{C}_j} c$ 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

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

Figure 4.4: Part of Γ_{OM}

the terms g_k. To realize `ViewMeal.ain` for the context (s, e, i), for example, we ask the four inhabitation questions

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

Figure 4.3c depicts the result.

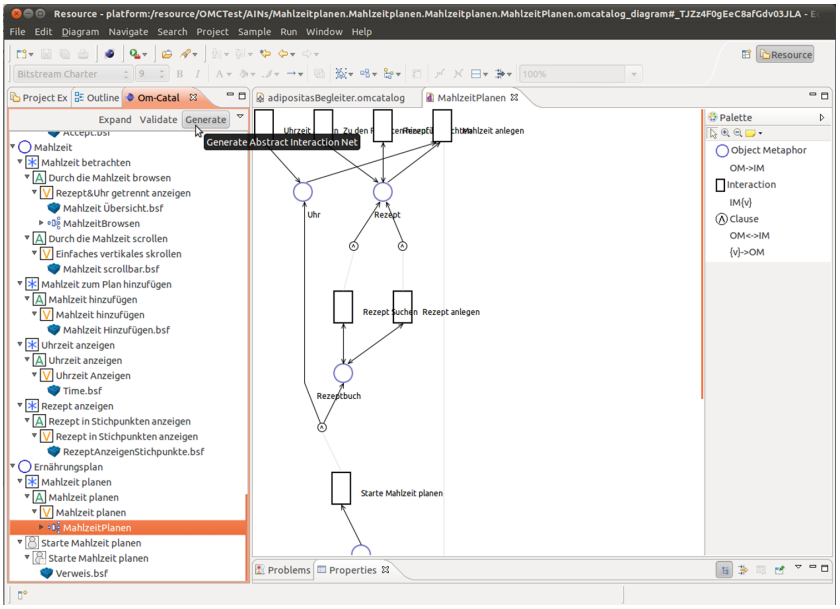
The following section discusses (part of) a realization of this approach towards synthesizing GUIs.

5 IMPLEMENTATION

We presented a prototypical Prolog-implementation of the ATM shown in Figure 2.2 deciding inhabitation in BCL_0 [4]. It uses SWI-Prolog [22] and is based on a standard representation of alternation in logic programming [20]. 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 [8, 17] based on Eclipse [6] 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 5.1a) 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 5.1b contains an enlarged extract of the repository displayed in Figure 5.1a. It is a direct manifestation of the repository from which components are drawn for the synthesis of a GUI.

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.



(a) Repository with example AIN

(b) Extract of repository

Figure 5.1: GUI for synthesis-framework

6 RELATED WORK

We are not aware of previous work very directly related to the approach proposed here. Our approach is broadly related in spirit to adaptation synthesis via proof counting [9, 21], where semantic specifications at the type level are combined with proof search in a specialized proof system. In contrast, we consider composition synthesis, and our logic (bounded combinatory logic with intersection types) is different. The presence of intersection together with k-bounded polymorphism yields enormous theoretical expressive power (simulation of alternating space bounded Turing machines) and complexity (nonelementary recursive when the bound k is a parameter). Problems of synthesis from component libraries have been investigated within temporal logic and automata theory [12]. 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. Furthermore, for better scalability and functionality we plan to reimplement the algorithm, using the .NET-framework [1], which will allow for a much greater flexibility than the Prolog-based implementation. Experimental application and evaluation of the ideas described here will be pursued in a number of different areas, including GUI synthesis, robotics, and planning. Finally, as mentioned above, a comparison to synthesis problems framed in temporal logics could be interesting.

BIBLIOGRAPHY

[1] .net development. http://msdn.microsoft.com/en-us/library/ff361664.aspx.

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

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

[4] Düdder, B., Martens, M., and Rehof, J. Prototype implementation of an inhabitation algorithm for fcl$(\cap, \leq)$. Presentation at Types 2011 in Bergen, Norway, September 2011.

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

[6] Eclipse.org. Eclipse indigo (3.7) documentation.

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

[8] Garbe, O. Synthese von benutzerschnittstellen mit einem typinhabitationsalgorithmus. Diploma thesis, Technical University of Dortmund, March 2012.

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

[10] Königsmann, T. *Compositional Modelling Ansatz zur Benutzerschnittstellengenerierung am Beispiel telemedizinischer Anwendungen*. PhD thesis, Technical University of Dortmund, 2011.

[11] Königsmann, T., and Kriebel, R. Digitale gesundheitsbegleiter am beispiel der adipositas-nachsorge. In *Proceedings of AAL 2008* (2008), Verband der Elektrotechnik, Elektronik, Informationstechnik, VDE-Verlag.

[12] Lustig, Y., and Vardi, M. Y. Synthesis from component libraries. In *FOSSACS* (2009), vol. 5504 of *LNCS*, Springer, pp. 395–409.

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

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

[15] Rehof, J., and Urzyczyn, P. Finite combinatory logic with intersection types. In *TLCA* (2011), vol. 6690 of *Lecture Notes in Computer Science*, Springer, pp. 169–183.

[16] Rehof, J., and Urzyczyn, P. The complexity of inhabitation with explicit intersection. In *Kozen Festschrift* (2012), R. L. Constable and A. Silva, Eds., vol. LNCS 7230, pp. 256–270.

[17] Reinke, E. Konzeption und entwicklung eines tools zur modellierung von user-interface-spezifikationsbausteinen im rahmen des "compositional modeling"-ansatzes. Diploma thesis, Technical University of Dortmund, 2011.

[18] Salvati, S. Recognizability in the simply typed lambda-calculus. In *WoLLIC* (2009), H. Ono, M. Kanazawa, and R. J. G. B. de Queiroz, Eds., vol. 5514 of *LNCS*, Springer, pp. 48–60.

[19] Salvati, S., Manzonetto, G., Gehrke, M., and Barendregt, H. Loader and urzyczyn are logically related. In *ICALP 12, Automata, Languages, and Programming - 39th International Colloquium, Warwick, UK* (2012), vol. 7392 of *LNCS*, Springer, pp. 364–376.

[20] Shapiro, E. Y. Alternation and the computational complexity of logic programs. *J. Log. Program. 1*, 1 (1984), 19–33.

[21] Wells, J. B., and Yakobowski, B. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR 2004* (2005), S. Etalle, Ed., vol. 3573 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 262–277.

[22] Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. Swi-prolog. *Computing Research Repository abs/1011.5332* (2010).

Forschungsberichte
der Fakultät für Informatik
der Technischen Universität Dortmund

ISSN 0933-6192

Anforderungen an:
Dekanat Informatik | TU Dortmund
D-44221 Dortmund