A Type-Level Approach To Component Prototyping

Luís Barbosa
DI-CCTC, Univ. do Minho
Portugal
lsb@di.uminho.pt

Jácome Cunha[*]
DI-CCTC, Univ. do Minho
Portugal
jacome@di.uminho.pt

Joost Visser
Software Improvement Group
The Netherlands
j.visser@sig.nl

ABSTRACT

Algebraic theories for modeling components and their interactions offer abstraction over the specifics of component states and interfaces. For example, such theories deal with forms of sequential composition of two components in a manner independent of the type of data stored in the states of the components, and independent of the number and types of methods offered by the interfaces of the combinators. General purpose programming languages do not offer this level of abstraction, which implies that a gap must be bridged when turning component models into implementations.

In this paper, we present an approach to prototyping of component-based systems that employs so-called type-level programming (or compile-time computation) to bridge the gap between abstract component models and their type-safe implementation in a functional programming language. We demonstrate our approach using Barbosa's model of components as generalized Mealy machines. For this model, we develop a combinator library in HASKELL, which uses type-level programming with two effects. Firstly, wiring between components is computed during compilation. Secondly, the well-formedness of the component compositions is guarded by HASKELL's strong type system.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Automata*

General Terms

Design, Languages, Theory, Verification

Keywords

HASKELL, Type-level programming, Mealy machine, Coalgebra, Combinator library

[*]Supported by the Fundação para a Ciência e a Tecnologia, Portugal, under grant number SFRH/BD/30231/2006.

1. INTRODUCTION

Over the last decade component-based software development [38, 39] emerged as a promising paradigm to deal with the ever increasing need for mastering complexity in software design, evolution and reuse. As a paradigm it retains from object-orientation the basic principle of encapsulation of data and code, but shifts the emphasis from (class) inheritance to (object) composition. This shift avoids interference between the inheritance and encapsulation, paving the way to a development methodology based on assembly of third-party components.

From the outset, the basic motivation has been to replace conventional programming by the composition and configuration of reusable *off–the–shelf* units, often regarded as *'abstractions with plugs'* [29]. In this sense, a *component* is a 'black-box' entity which both provides and requires services, encapsulated through a public interface. Connections are established by drawing *wires*, corresponding to some sort of interfacing code.

In practice, however, software components do not fit together as easily as *Lego* pieces. This motivated new research questions concerning component adaptation, wrapping, composition and interaction. A number of answers has been formulated to such questions, often from disparate points of view, either technological, methodological or foundational. So far, none of these approaches has emerged as the final or predominate answer to component composition.

In particular, foundational approaches to component composition face the challenge of transposing proposed mathematical descriptions of components and their interactions into executable programs or prototypes thereof. This requires the encoding of abstract mathematical structures and operators into the concrete constructs and libraries of general-purpose programming languages. In this paper, we pick up this challenge for a particular mathematical component theory, *i.e.* Barbosa's coalgebraic model of components as generalized Mealy machines [5, 3], and for a particular programming language, *i.e.* the strongly-typed functional programming language HASKELL [16].

A Mealy machine is a finite state transducer that, upon accepting an input signal, modifies its internal state and generates an output signal [25], which can be captured by the following function type:

$$U \times I \longrightarrow U \times O$$

where I and O are the input and output alphabet, and U is a finite set of states. Generalizing this idea, components can be modeled by curried functions of the following form:

$$U \longrightarrow I \longrightarrow \mathsf{B}(U \times O)$$

where B is a monad that captures behavioral models such as partiality or non-determinism. These functions can be recognized as coalgebras:

$$U \longrightarrow T_{I,O}U$$

where

$$T_{I,O}X = I \longrightarrow \mathsf{B}(X \times O)$$

Here, $T_{I,O}$ is a datatype transformer (formally a *functor*). The input and output alphabets can be recognized as n-ary labeled sums, where the label is a signal or message, and the summand represents the parameters of that message. The *interface* of a component hides the encapsulated state, and has a signature of the form $I \longrightarrow O$. Basic components can then be created from n-ary products of state-modifying functions. Subsequently, basic components can be composed into complex components using component composition operators, or component combinators. For example, the binary operator **;** for sequential composition accepts input of type:

$$(U_1 \longrightarrow T_{I,L}U_1) \times (U_2 \longrightarrow T_{L,O}U_2)$$

to produce a new component over the combined state space of type:

$$(U_1 \times U_2) \longrightarrow T_{I,O}(U_1 \times U_2)$$

where L is both the output language for the first component, and the input language for the second. In Section 2, the theory of components as generalized Mealy machines is recapitulated in more detail.

Haskell is a non-strict functional programming language with a rich system of strong types. Haskell features higher-order functions, parametric polymorphism, and ad-hoc polymorphism or overloading through its notion of type classes. The standard libraries of the language include support for a variety of monads, and special syntax is provided to support monadic programming. A commonly used language extension, *viz* multi-parameter type-classes with functional dependencies, allows static (compile-time) computations to be expressed by logic programming on the level of types [13, 24]. This emergent capability has been exploited for instance to model n-ary products, extensible records [19], functions with variable length argument lists [18], and relational databases [37]. As we will explain, this mix of features allows a faithful transposition of the chosen component model. More concretely, we will present a Haskell library of combinators that supports the construction of executable prototypes of component compositions.

The paper is structured as follows. In Section 2, we present as motivating example the component-based specification of a simple electronic voting system. By means of this example, we also provide a detailed recapitulation of the coalgebraic model of components as generalized Mealy machines. In Section 3, we provide background on the Haskell language in general, and about the technique of type-level programming in particular. Our combinator library for component-based programming is presented in Section 4, showing in detail how the various ingredients of the theory are transposed. In Section 5, we revisit the motivating example, to show how our library can be used to turn it into an executable prototype. Finally, Section 6 discusses related work and concludes.

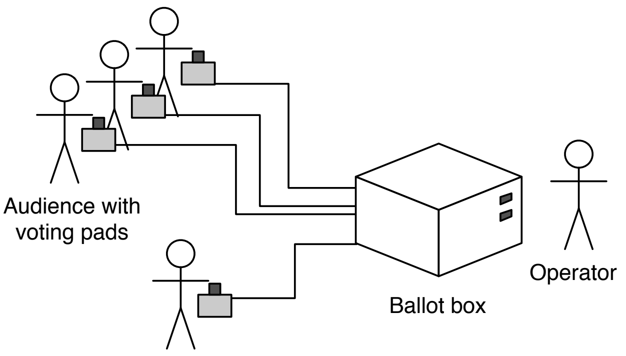


Figure 1: A simple voting system, where the audience, for example of a TV show, can press voting pads connected to a central ballot box. The operator initiates the ballot box with the number of votes that must be collected. When sufficient votes are in, the status indicator of the ballot box changes.

2. A THEORY OF COMPONENTS

This section recapitulates a calculus of state-based components framed as generalized Mealy machines [5, 3]. Our exposition is driven by a small example, used throughout the paper, which illustrates both the sort of models we are interested in and a number of composition operators.

The example is a highly simplified voting system, illustrated in Figure 1. The system consists of a central unit to collect the votes, *i.e.* a *ballot box* of some sort. The votes are collected via single-button *voting pads*, in the hands of the voters. An operator initiates the voting process by resetting the ballot box with the number of votes that must be collected. The voters can press their voting pad to register a vote. When sufficient votes are in, a status indicator on the ballot box changes. A common use of such a system can be found in processing units for electronic opinion polls, as in some television shows. Presumably, when enough members of the audience have expressed their discontent by lodging a vote, the current performers are sent off stage. A similar system, however, can be used to count inputs from a number of sensors in, *e.g.*, an industrial plant. Typically, such sensors emit a number of stimuli before terminating.

In the course of this section, we will explain how the various components of the system, *i.e.* the voting pads and ballot box, are modeled as generalized Mealy machines, and how their wiring is specified using component combinators. The components themselves are explained in Sections 2.1, 2.2, and 2.3. The composition operators are presented in Section 2.4. Finally, the synthesis of the components, using the operators, into the complete voting system is given in Section 2.5.

2.1 Generalized Mealy machines

As indicated briefly in the introduction, software components can be modeled as generalized Mealy machines. Driven by our example, we will build up this view of components in a step-wise fashion.

Let us first consider the state-changing functions that lie at the heart of our two types of components: the voting pad and the ballot box.

In our example each voter has a given number of votes. The voting pad maintains, then, a natural number $n \in \mathbb{N}$ as state information, indicating the number of stimuli remaining to be emitted by the device. This stimuli is here represented by the singleton (or unit) datatype $\mathbf{1}$. A single operation is available for changing this state:

$$\mathsf{emit} : \mathbb{N} \times \mathbf{1} \longrightarrow (\mathbb{N} \times \mathbf{1}) + \mathbf{1}$$

Here, the + operator indicates the sum type, or disjoint union. Pressing the voting pad button amounts to invoking the emit function on the internal state. The presence of the *maybe* monad $(X + \mathbf{1})$ in the result type indicates that the overall behavior of this component is *partial*: when the allowed number of votes has run out $(n = 0)$, the vote emission operation fails.

At the heart of the ballot box, on the other hand, we find two operations over a state space of type $\mathbb{N}$:

$$\mathsf{reset} : \mathbb{N} \times \mathbb{N} \longrightarrow \mathsf{B}(\mathbb{N} \times \mathbf{1})$$
$$\mathsf{vote} : \mathbb{N} \times \mathbf{1} \longrightarrow \mathsf{B}(\mathbb{N} \times \mathbf{2})$$

The reset operation sets the minimum number of votes required to report success, which means that the internal state (first argument of type $\mathbb{N}$) is overwritten with the argument of the operation (second argument of type $\mathbb{N}$). The vote operation counts an individual vote, by decreasing the state value (argument of type $\mathbb{N}$). The output of action vote is a boolean flag (type $\mathbf{2}$) indicating whether all votes requested to terminate the voting process have been received. The behavior monad B wrappes the component's result (explained just bellow).

These various state-changing functions can in fact be recognized as special cases of a single general pattern:

$$\mathsf{f} : U \times I \longrightarrow \mathsf{B}(U \times O) \qquad (1)$$

In this pattern, U is the type of the internal state, I and O are the types of the input and output signals, respectively, and B is a strong monad[1].

The monad parameter captures the *behavioral model* of the state changing computation. When B is the identity monad, the general pattern reduces to the special case of Mealy machines [25], whose behavior is simply to compute an output and a new state. The *maybe* monad, corresponding to the behavioral model of the emit operation above, captures *partial* computations. As further behavioral models, one can also think of components behaving within a certain degree of non-determinism or following a probability distribution. Thus, in general computation of an action will not simply produce an output and a continuation state, but a B-structure of such pairs. The monadic structure provides tools to handle such computations. Unit (η) and multiplication (μ), provide, respectively, a value embedding and a 'flatten' operation to reduce nested behavioral effects. Strength, either in its right (τ_r) or left (τ_l) version, cater for context information. Thus, the presence of B as a parameter in the definition allows instantiation with different kinds of behavior models and justifies the qualifier 'generalized'.

We prefer to write the general pattern (1) in its curried form, where exponent notion is used for one of the function types [2]:

$$\mathsf{f} : U \longrightarrow \mathsf{B}(U \times O)^I$$

After currying, it becomes evident that our state-changing functions can be recognized as *coalgebras* [34] of the form:

$$U \longrightarrow \mathsf{T}_{I,O}\, U$$

where

$$\mathsf{T}_{I,O} X \;= \mathsf{B}(X \times O)^I$$

is the coalgebra's datatype transformer, or *functor.*

2.2 Encapsulation

An alternative, 'black box' view hides both state and behavioral monad information from the components' environments and regards each operation as a pair of input/output ports. Such a 'port' signature of, *e.g.*, the emit operation is then given by:

$$\mathsf{emit} : \mathbf{1} \longrightarrow \mathbf{1}$$

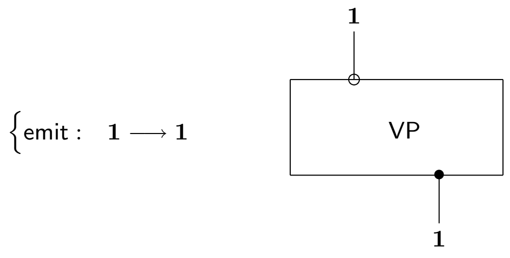
Note that we have omitted from emit's signature both the state argument and the *maybe* monad in the output. Figure 2 illustrates this interface view of the voting pad VP. In the diagram the input (respectively, output) interface is represented by the sum of its port types (here the sum has only one element) and depicted as an empty (respectively, full) circle. Such a representation makes explicit the elementary interface of the voting pad component: trivial input and output means that no special data is exchanged by this component — simply, a button is pushed (on input) and a led lights up (on output).

Similarly, the port signature for vote, on the ballot box BB, is:

$$\mathsf{vote} : \mathbf{1} \longrightarrow \mathbf{2}$$

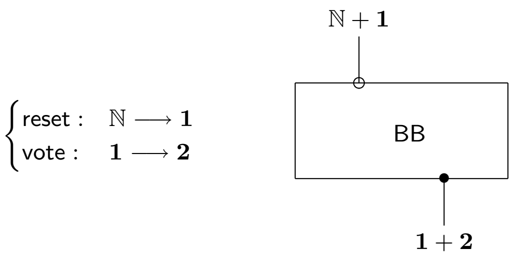
[1]A *strong monad* is a monad $\langle \mathsf{B}, \eta, \mu \rangle$ where B is a strong functor and both η and μ are strong natural transformations [21]. B being strong means there exist natural transformations $\mathsf{T}(\mathsf{Id} \times -) : \mathsf{T} \times - \Longleftarrow \mathsf{T} \times -$ and $\mathsf{T}(- \times \mathsf{Id}) : - \times \mathsf{T} \Longleftarrow - \times \mathsf{T}$, called the right and left strength, respectively, subject to certain conditions. Their effect is to *distribute* the free variable values in the context "$-$" along functor B.

[2]We resort to the exponent notation X^I for functional dependency, standard in mathematics, rather than the equivalent $I \rightarrow X$, more familiar in computing, in order to emphasize the dependency of the possible observations X from the input I.



$\left\{\mathsf{emit}: \ \mathbf{1} \longrightarrow \mathbf{1}\right.$
VP
1
1

Figure 2: The interface of the voting pad VP.



$\mathsf{reset}: \ \mathbb{N} \longrightarrow \mathbf{1}$
$\mathsf{vote}: \ \mathbf{1} \longrightarrow \mathbf{2}$
BB
$\mathbb{N}+\mathbf{1}$
$\mathbf{1}+\mathbf{2}$

Figure 3: The interface of the ballot box BB.

retaining only the output value. For reset one gets:

$$\mathsf{reset}: \mathbb{N} \longrightarrow \mathbf{1}$$

meaning that an external argument is required on activation but no visible output is produced, except for a trivial indication of successful termination. This is illustrated in Figure 3. The combined input type $\mathbb{N}+\mathbf{1}$ models the choice of two functionalities, of which only one takes input of type $\mathbb{N}$.

In general, then, the interface of a component takes the form of a function type between two n-ary sum types:

$$p : \Sigma_n I_i \longrightarrow \Sigma_n O_i$$

The two sum types are of equal length, and the respective summands of each type correspond to the input and output signals of the various operations supported by the component.

2.3 Component specifications

Let us now turn to the actual behavior of the voting pad and ballot box components. A behavioral specification of a software component can be given by a pointed coalgebra[3]:

$$p : I \longrightarrow O \ = \ \langle u_p \in U_p, \overline{a}_p : U_p \longrightarrow \mathsf{B}(U_p \times O)^I \rangle$$

where u_p is the initial state, often referred to as the *seed* of the component computation, and $\overline{a}_p$ is the curried version of a state transition function a_p, capturing the coalgebra dynamics.

For example, the voting pad is specified as follows:

$$\mathsf{VP} : \mathbf{1} \longrightarrow \mathbf{1} \quad = \quad \langle n \in \mathbb{N}, \overline{\mathsf{emit}} \rangle$$

[3]This kind of coalgebras have a seed value, *i.e.* a value which acts as an initial state for the underlying transition system.

with[4]:

$$\mathsf{emit}\ \langle n, * \rangle \ = (n \neq 0 \ \rightarrow \ \iota_1\ \langle n-1, * \rangle,\ \iota_2\ *)$$

On the other hand, the ballot box is modelled as:

$$\mathsf{BB} : \mathbb{N}+\mathbf{1} \longrightarrow \mathbf{1}+\mathbf{2} \quad = \quad \langle n \in \mathbb{N}, \overline{a}_{\mathsf{BB}} \rangle$$

where dynamics $\overline{a}_{\mathsf{BB}} = \mathsf{reset} \oplus \mathsf{vote}$ is based on actions reset and vote:

$$\begin{aligned} \mathsf{reset}\ \langle u, m \rangle \ &= \iota_1\ \langle m, * \rangle \\ \mathsf{vote}\ \langle u, * \rangle \ &= \iota_1\ \langle u-1, u=1 \rangle \end{aligned}$$

Their combination $\mathsf{reset} \oplus \mathsf{vote}$ is specified as follows:

$$\begin{aligned} & U \times (I_1 + I_2) \\ & \xrightarrow{\mathsf{dr}} (U \times I_1) + (U \times I_2) \\ & \xrightarrow{\mathsf{reset}+\mathsf{vote}} \mathsf{B}(U \times O_1) + \mathsf{B}(U \times O_2) \\ & \xrightarrow{[\mathsf{B}(\mathsf{id}\times\iota_1),\mathsf{B}(\mathsf{id}\times\iota_2)]} \mathsf{B}(U \times (O_1 + O_2)) \end{aligned}$$

In the first step, the state U is distributed over the summands of the input language, after which either reset or vote is applied. The resulting alternative monadic computations are combined into a single monadic computation of a new state and the sum of possible output signals.

In general, a component is specified using an initial state and an n-tuple of monadic state-changing functions:

$$p : \Sigma_n I_i \longrightarrow \Sigma_n O_i \ = \ \langle u_p \in U_p, \overline{\oplus_n(\Pi_n f_i)} \rangle$$

where:

$$f_i : U_p \times I_i \longrightarrow \mathsf{B}(U_p \times O_i)$$

The operator $\oplus_n$ is the monadic, n-ary generalization of $\oplus$.

Thus, we have arrived at a general recipe for modelling state-based components: input-output interfaces, an encapsulated state and 'black-box' behavior 'packed' as a concrete pointed coalgebra. For a given initial value of the state space, a corresponding 'process', or *behavior*, arises by computing its coinductive extension[5]. As we will see ahead, we will use type-level programming to capture the various n-ary type constructors and operators that are involved in these component specifications.

2.4 An algebra of components

From individual components, we wish to construct larger systems in a compositional manner.

To this end, the component calculus offers an algebra of component combinators such as *pipeline* (;), and three tensors that capture *external choice* ($\boxplus$) as well as *parallel* ($\boxtimes$) and *concurrent* ($\circledast$) composition. Generalized *interaction* is catered through a 'feedback' combinator, called *hook* ($\urcorner$), connecting a specified subset of outputs to a subset of inputs

[4]Recall conditional construction ($p \ \rightarrow \ f,\ g$), whose meaning is if p then f else g. $*$ is the only habitant of the $\mathbf{1}$ datatype.
[5]The 'black-box' characterization of software components favors an *observational* semantics: any two internal configurations should be considered identical whenever indistinguishable by observation. This is nicely captured by taking coalgebraic theory as the semantic framework for a component's algebra. For details see [3, 4].

of the same component. This allows arbitrary communication between components to be achieved by first aggregating them via one of the tensors and then selecting the input and output points to be connected by *hook*. Finally component adaptation is captured by a *wrapping* combinator ($[\,]$).

The *basic* components on which these combinators operate can be specified on the basis of an initial state and state-changing functions, as explained above. Alternatively, an operation of *function lifting* ($\ulcorner\ \urcorner$) allows arbitrary functions to be promoted to (state-less) components, after which they can likewise be composed.

We shall restrict ourselves to the presentation of just a few combinators, emphasizing the ones used in the voting system example. The reader is referred elsewhere [3, 4] for the formal definition of these combinators as well as the various *laws* they obey.

2.4.1 Function lifting

A simple mechanism can be applied to promote any function $f : A \longrightarrow B$ to a component with trivial state $\mathbf{1}$:

$$\ulcorner f \urcorner : A \longrightarrow B \;=\; \langle * \in \mathbf{1}, \overline{a}_{\ulcorner f \urcorner} \rangle$$

where

$$a_{\ulcorner f \urcorner} \;= \mathbf{1} \times A \xrightarrow{\mathsf{id} \times f} \mathbf{1} \times B \xrightarrow{\eta_{(\mathbf{1} \times B)}} \mathsf{B}(\mathbf{1} \times B)$$

Such state-less functions are typically used for interface adaptation.

Various simple components arise by lifting standard elementary functions. For example, the lift of the *null function*, *i.e.* the identity on the empty set, plays the role of an *inert* component, unable to react to the outside world:

$$\mathsf{nil} : \emptyset \longrightarrow \emptyset \;=\; \ulcorner \mathsf{id}_\emptyset \urcorner$$

A somewhat dual role is played by the idling component:

$$\mathsf{idle} : \mathbf{1} \longrightarrow \mathbf{1} \;=\; \ulcorner \mathsf{id}_\mathbf{1} \urcorner$$

Note that idle will propagate an unstructured stimulus (*e.g.*, pushing a button) leading to a (similarly) unstructured reaction (*e.g.*, exciting a led).

A general identity-lifting operator is defined as follows:

$$\mathsf{copy}_X : X \longrightarrow X \;=\; \ulcorner \mathsf{id}_X \urcorner$$

A copy component copy_X simply repeats its input values on its output port.

2.4.2 Wrapping

An alternative way to introduce functions in the calculus is provided by the *wrapping* combinator, which is reminiscent of the *renaming* connective found in process calculi (*e.g.*, [27]). Let $p : I \longrightarrow O$ be a component and consider functions $f : I' \longrightarrow I$ and $g : O \longrightarrow O'$. Component p wrapped by f and g, denoted by $p[f, g]$ and typed as $I' \longrightarrow O'$, is defined by input pre-composition with f and output post-composition with g. Formally:

$$\begin{aligned} a_{p[f,g]} &= U_p \times I' \\ &\xrightarrow{\mathsf{id} \times f} U_p \times I \\ &\xrightarrow{a_p} \mathsf{B}(U_p \times O) \\ &\xrightarrow{\mathsf{B}(\mathsf{id} \times g)} \mathsf{B}(U_p \times O') \end{aligned}$$

As expected, the following properties hold:

$$\begin{aligned} p[f, g] &\sim \ulcorner f \urcorner \,;\, p \,;\, \ulcorner g \urcorner \\ (p[f, g])[f', g'] &\sim p[f \cdot f', g' \cdot g] \end{aligned}$$

Here, $\sim$ denotes *bisimilarity*, the meaningful notion of equivalence for state-based components. Thus, wrapping a component with a function is bisimular to pipeline composition with the component that results from lifting the function. Also, multiple wrappings are bisimular to a single wrapping with composed functions.

2.4.3 Pipeline

The *pipeline* aggregation of two components p and q is defined as a new component over the product of the two state spaces: the output of p is passed to q in a monadic way. Notice that all definitions (and laws) are parametric on monad B. Formally:

$$p \,;\, q : I_p \longrightarrow O_q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p;q} \rangle$$

where $a_{p;q} : U_p \times U_q \times I_p \longrightarrow \mathsf{B}(U_p \times U_q \times O_q)$ is detailed as follows:

$$\begin{aligned} & U_p \times U_q \times I_p \\ & \xrightarrow{\cong} U_p \times I_p \times U_q \xrightarrow{a_p \times \mathsf{id}} \mathsf{B}(U_p \times K) \times U_q \\ & \xrightarrow{\tau_r} \mathsf{B}(U_p \times K \times U_q) \xrightarrow{\cong} \mathsf{B}(U_p \times (U_q \times K)) \\ & \xrightarrow{\mathsf{B}(\mathsf{id} \times a_q)} \mathsf{B}(U_p \times \mathsf{B}(U_q \times O_p)) \\ & \xrightarrow{\mathsf{B}\tau_l} \mathsf{BB}(U_p \times (U_q \times O_p)) \\ & \xrightarrow{\cong} \mathsf{BB}(U_p \times U_q \times O_p) \xrightarrow{\mu} \mathsf{B}(U_p \times U_q \times O_p) \end{aligned}$$

Here, the intermediate language $K = I_q = O_p$. Note that the form of interaction underlying this combinator can be made partial, in the sense that only *part of* the output of one component needs to be fed as input to the other. In this case, $K = I_q \subseteq O_p$, the p output must be wrapped with the morphism which makes the inclusion.

Pipeline composition has a monoidal structure up to bisimulation. That is, for appropriately typed components p, q and r:

$$\begin{aligned} \mathsf{copy}_I \,;\, p \;\sim\; p \;\sim\; p \,;\, \mathsf{copy}_O \\ (p \,;\, q) \,;\, r \;\sim\; p \,;\, (q \,;\, r) \end{aligned}$$

where copy_X is the lifting of the monadic unit, as explained above.

2.4.4 Hook

The hook operator ($\urcorner_Z$) creates a feedback loop from the output of a component back to its input. The hook operator must be applied to components of type $I + Z \longrightarrow O + Z$, where Z is the type of the feedback wire. The type of the resulting component is also $I + Z \longrightarrow O + Z$. Figure 4

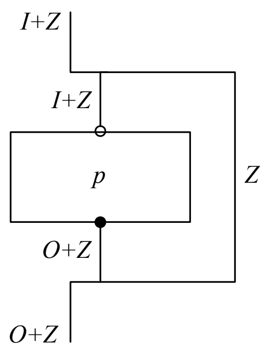


Figure 4: The hook combinator.

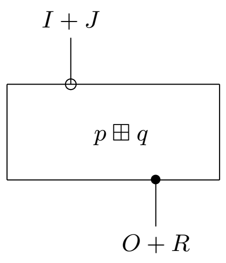


Figure 5: External choice combinator.

illustrates this. Operationally, the hooked component reacts to input of type $I + Z$ by producing output of type O, in which case it terminates, or of type Z, in which case the output is fed back into the input port of type Z. The formal definition is as follows:

$$p \Lsh_Z : I + Z \longrightarrow O + Z \;=\; \langle u_p \in U_p, \overline{a}_{p \Lsh_Z} \rangle$$

where $a_{p \Lsh_Z} : U_p \times (I + Z) \longrightarrow \mathsf{B}(U_p \times (O + Z))$ is detailed by:

$$\begin{aligned}
& U_p \times (I + Z) \\
& \xrightarrow{a_p} \mathsf{B}(U_p \times (O + Z)) \xrightarrow{\mathsf{Bdr}} \mathsf{B}(U_p \times O + U_p \times Z) \\
& \xrightarrow{\mathsf{B}(\mathsf{id} \times \iota_1 + \mathsf{id} \times \iota_2)} \mathsf{B}(U_p \times (O + Z) + U_p \times (I + Z)) \\
& \xrightarrow{\mathsf{B}(\eta + a_p)} \mathsf{B}(\mathsf{B}(U_p \times (O + Z)) + \mathsf{B}(U_p \times (O + Z))) \\
& \xrightarrow{\mathsf{B}\triangledown} \mathsf{BB}(U_p \times (O + Z)) \xrightarrow{\mu} \mathsf{B}(U_p \times (O + Z))
\end{aligned}$$

Typically, the hook operator is applied to the parallel composition of various components, where the effect of the hook operator is to feed the output of one component into the input of another.

2.4.5 Tensors

Besides 'pipeline' composition, components can be aggregated in a number of different ways, captured by tensor products corresponding to *choice*, *parallel* and *concurrent* composition. We shall only focus here *external choice* which, for $p : I \longrightarrow O$ and $q : J \longrightarrow R$, is depicted in Figure 5. When interacting with $p \boxplus q$, the environment is allowed to choose either to input a value of type I or one of type J, triggering the corresponding component (p or q, respectively) and producing output. Formally:

$$p \boxplus q \;=\; \langle \langle u_p, u_q \rangle \in U_p \times U_q, \overline{a}_{p \boxplus q} \rangle$$

where $a_{p \boxplus q}$ is detailed as follows:

$$\begin{aligned}
& U_p \times U_q \times (I + J) \\
& \xrightarrow{\cong} U_p \times I \times U_q + U_p \times (U_q \times J) \\
& \xrightarrow{a_p \times \mathsf{id} + \mathsf{id} \times a_q} \mathsf{B}\,(U_p \times O) \times U_q + U_p \times \mathsf{B}\,(U_q \times R) \\
& \xrightarrow{\tau_r + \tau_l} \mathsf{B}\,(U_p \times O \times U_q) + \mathsf{B}\,(U_p \times (U_q \times R)) \\
& \xrightarrow{\cong} \mathsf{B}\,(U_p \times U_q \times O) + \mathsf{B}\,(U_p \times U_q \times R) \\
& \xrightarrow{[\mathsf{B}\,(\mathsf{id} \times \iota_1), \mathsf{B}\,(\mathsf{id} \times \iota_2)]} \mathsf{B}\,(U_p \times U_q \times (O + R))
\end{aligned}$$

The combinator satisfies a number of laws useful to reasoning about component-oriented design [3]. For example:

$$\begin{gathered}
(p \boxplus p')\,;(q \boxplus q') \sim (p\,;q) \boxplus (p'\,;q') \\
\mathsf{copy}_{K \boxplus K'} \sim \mathsf{copy}_K \boxplus \mathsf{copy}_{K'} \\
(p \boxplus q) \boxplus r \sim (p \boxplus (q \boxplus r))[\mathsf{a}_+, \mathsf{a}_+{}^\circ] \\
\mathsf{nil} \boxplus p \;\sim\; p[\mathsf{r}_+, \mathsf{r}_+{}^\circ] \text{ and } p \boxplus \mathsf{nil} \;\sim\; p[\mathsf{l}_+, \mathsf{l}_+{}^\circ] \\
p \boxplus q \sim (q \boxplus p)[\mathsf{s}_+, \mathsf{s}_+]
\end{gathered}$$

Notice the use of wrapping, in the last few laws, to assure the input/output interfaces of both sides of the equality are made compatible. Note that s_+ is the commutativity isomorphism for sum, a_+ is the morphism which witnesses the sum associative law, r_+ is the function that transforms $\mathbf{1} + A$ into A and l_+ transforms $A + \mathbf{1}$ into A.

The other two tensors are *parallel* composition $p \boxtimes q : I \times J \longrightarrow O \times R$ and *concurrent* aggregation $p \boxast q : I + J + I \times J \longrightarrow O + R + O \times R$. Parallel composition corresponds to a synchronous product: both components are executed simultaneously when triggered by a pair of legal input values. Note, however, that the behavior effect, captured by monad B, propagates. For example, if B captures component failure and one of the arguments fails, the product will fail as well. Concurrent aggregation combines *choice* and *parallel*, in the sense that p and q can be executed independently or jointly, depending on the input supplied.

We can generalize the binary tensors for choice and parallel composition to n-ary versions with the following types:

$$\begin{gathered}
\boxplus_n p_i : \Sigma_n I_i \longrightarrow \Sigma_n O_n \\
\boxtimes_n p_i : \Pi_n I_i \longrightarrow \Pi_n O_i
\end{gathered}$$

An example of their use will follow below.

2.5 Component composition

The purpose of this section is to illustrate how new components can be built from existing ones, relying on the calculus sketched above. The example is the construction of the *voting system* of Figure 1 out of the *ballot box* component and n *voting pad* components, specified in Section 2.3.

An n-voting system is assembled by aggregating n voting pads and connecting their outputs to the ballot box with an n-diagonal wire, as illustrated in Figure 6.

Recall that a *codiagonal* is a function $\triangledown : A + A \longrightarrow A$ defined as the *either* of two identities, *i.e.* $\triangledown = [\mathsf{id}, \mathsf{id}]$. This

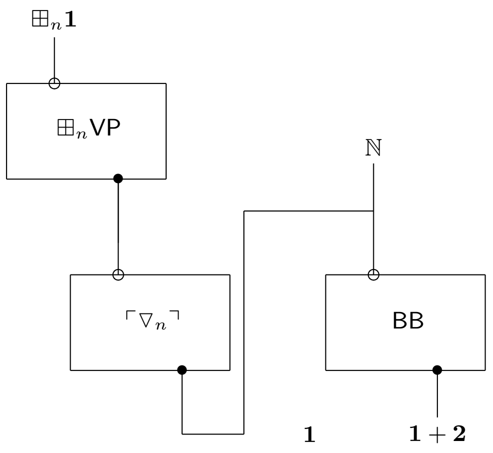


Figure 6: Assembling the n voting system.

binary operator can be generalized to an n-ary operator $\nabla_n : \Sigma_n A \longrightarrow A$. When this function is lifted, we obtain a component that concentrates n wires of type A into a single wire. Thus, we begin component composition with:

$$\mathsf{S}_n \;=\; (\boxplus_n \mathsf{VP} \,;\, \ulcorner\nabla_n\urcorner) \boxplus \mathsf{BB}$$

which is typed as $\mathsf{S}_n : \boxplus_n \mathbf{1} + (\mathbb{N} + \mathbf{1}) \longrightarrow \mathbf{1} + (\mathbf{1} + \mathbf{2})$. Thus, we now have a system with two components operating in parallel: the combination of n voting pads with the n-codiagonal consumes input of type $\boxplus_n \mathbf{1}$ and produces output of type $\mathbf{1}$; and the ballot box which consumes $\mathbb{N} + \mathbf{1}$ and produces $\mathbf{1} + \mathbf{2}$.

To make these parallel combinators interact, we need to use the *hook* combinator. To apply *hook*, however, S_n has to be wrapped to exhibit the hooked type in the correct position. For this, $\mathsf{S}_n[\mathsf{a}_+, \mathsf{s}_+]$ has the right type: $(\boxplus_n \mathbf{1} + \mathbb{N}) + \mathbf{1} \longrightarrow (\mathbf{1} + \mathbf{2}) + \mathbf{1}$. The voting system is, then, defined as

$$\mathsf{VS}_n \;=\; (((\boxplus_n \mathsf{VP} \,;\, \ulcorner\nabla_n\urcorner) \boxplus \mathsf{BB})\,[\mathsf{a}_+, \mathsf{s}_+])\, \Lsh_{\mathbf{1}}$$

which has type $\boxplus_n \mathbf{1} + \mathbb{N} \longrightarrow \mathbf{1} + \mathbf{2}$. Thus, the actions that are externally available correspond, respectively, to the act of emitting one of the n votes ($\mathbf{1}$) and resetting the voting threshold ($\mathbb{N}$).

Note that the architecture of VS_n is independent of the concrete specifications of components VP and BB. It remains valid, for example, if VP emits a value from an enumerated type (*e.g.*, the identifier of a candidate) and BB mantains, as its internal state, a bag of candidate identifiers to votes to be output when the all votes have been counted.

On the other hand, different composition patterns may be used to convey different specifications of their joint behavior. Note, for example, that in VS_n each vote is dealt separately. Replacing $\boxplus$ by $\circledast$ as the 'gluing' combinator of the voting pads, allows for the simultaneously counting of arbitrary chunks of votes. Eventually this suits reality better, as several voting pads may be activated at the same time.

3. TYPE-LEVEL PROGRAMMING

Before setting out on a transposition of the above algebraic theory of components to the functional language HASKELL (Section 4), we will provide the necessary background on this programming language. We will also briefly explain the technique of type-level programming that will prove to be instrumental in our encoding.

3.1 The Haskell type class

HASKELL is a non-strict, higher-order, typed functional programming language [16]. One of its most popular features is the possibility to group together functions with the same signature, over a certain data type, into what is called a *type class*. This declares an overloaded function with the declared signature. An instantiation mechanism provides particular implementations of such functions for particular types. For example, the following class declares a function *show* which transforms its argument into a string.

```
class Show a
  where
    show :: a → String
```

The following *instantiates* the class with the *Bool* data type:

```
instance Show Bool
  where
    show True = "T"
    show False = "F"
```

3.2 Type level programming

The HASKELL type class mechanism makes it possible to define functions over types. A type class with only one parameter (as the one in the example above) can be seen as a *predicate* on types. Similarly, multi-parameter classes encode *relations*. When a subset of the class parameters determines all the others, as indicated by the notation $\mid x\ y \rightarrow u\ v$ for functional dependencies among type parameters, type classes can be seen as functions on the level of types [13]. The following class can be seen as a function which computes type b from type a.

```
class Convert a b | a → b where
  convert :: a → b
```

The interesting part to note is that this computation is performed by the type checker, that is, at compile time. The arguments and results of type level functions are types that model values, sometimes designated as type-level values. The following example models natural numbers on the level of types:

```
data Zero
zero = ⊥ :: Zero
data Succ n
succ = ⊥ :: n → Succ n
class Natural n
instance Natural Zero
instance Natural n ⇒ Natural (Succ n)
```

Types *Zero* and *Succ* generate type-level values of type-level type *Natural*, which is a class. We can define now the sum operator over naturals:

```
class Add x y z | x y → z
  where
    add :: x → y → z
instance Add x Zero x
  where
    add x y = x
instance Add x y z ⇒ Add x (Succ y) (Succ z)
  where
    add x y = succ (add x (pred y))
pred :: Succ x → x
pred = ⊥
```

Note that class *Add* is a type-level function that models addition on naturals, whereas *add* adds naturals at the values level.

3.3 Heterogeneous collections

Kiselyov *et al.* use type level programming to model n-ary products, or heterogeneous lists as they call them [19]. Such lists are based on the following declarations:

```
data HNil = HNil
data HCons e l = HCons e l
class HList l
instance HList HNil
instance HList l ⇒ HList (HCons e l)
```

With data types *HNil* and *HCons* one may construct empty and non-empty heterogeneous lists, respectively.

The *HList* class establishes a well-formedness condition on heterogeneous lists, *i.e.* they are constructed with successive applications of the *HCons* constructure and end with *HNil*. *ex1* is an example of a well formed list with different type elements:

```
ex1 = HCons "foo" (HCons 2 (HCons True HNil))
```

The *HList* library has a number of useful operations available. For example, appending two heterogeneous lists:

```
class HAppend l l' l'' | l l' → l'' where
  hAppend :: l → l' → l''
```

Zipping two lists into a list of pairs and vice-versa:

```
class HZip l l' l'' | l l' → l'', l'' → l l' where
  hZip :: l → l' → l''
  hUnzip :: l'' → (l, l')
```

Infix operators to build lists are also provided (as synonyms) as syntax sugar:

```
type (:*:) e l = HCons e l
e .*. l = HCons e l
```

These infix operators on type and value level are closer to the mathematical notation for the n-ary products $(A_1 \times \ldots \times A_n)$ that can be modeled by them.

4. THE COMPONENT LIBRARY

We will now proceed to transpose the component calculus of Section 2 into a HASKELL combinator library for component prototyping. The library is based on type-level programming. We will both present our HASKELL component model (Section 4.2) and the suite of component combinators (Section 4.3). But first, we provide a key extension to the repertoire of type-level utility functions, *viz* a HASKELL encoding of n-ary sum types.

4.1 N-ary sums

As mentioned above, the HList library offers n-ary products, *i.e.* arbitrary length tuples. For our component library, we will also need an encoding of the dual concept of n-ary sums types. The following data type constructors form the basis of the encoding:

```
data HEither e l = HLeft e | HRight l
data HVoid
```

The basic idea is to construct sums types as left-associated applications of the *HEither* type constructor, terminated by *HVoid*, which represents the empty sum type. Thus, $A + B + C$ will be represented as:

```
HEither A (HEither B (HEither C HVoid))
```

The well-formedness of n-ary sums is guarded by the following class and instances:

```
class HSum s
instance HSum HVoid
instance HSum s ⇒ HSum (HEither e s)
```

Thus, the *HSum* class plays the same role for n-ary sums as the *HList* class for n-ary products.

Labeled sums types can be encoded as special cases of sum types where the summands are pairs of labels and element types. For example, the input language of our ballot box component has the following labeled sum type:

```
HEither (Vote, Either One Nat)
        (HEither (Reset, Either One Nat) HVoid)
```

The message of resetting the ballot box with argument value 20 is constructed by *HRight* (*HLeft* (⊥ :: *Reset*, *Right* 20)). Such construction of values of labeled sums can get quite cumbersome when sum types get larger. For more convenience, we have defined overloaded injection and selection functions:

```
class Sum l s x | l s → x
  where
    select :: l → s → Maybe x
    inject :: l → x → s
```

With these functions, we can construct the above value with:

```
vote20 = inject (⊥ :: Reset) (Right 20)
```

To retrieve the value 20 from the constructed sum type value, we would invoke *select* (⊥ :: *Reset*) *vote20*. The *select* operation is partial because it is possible that a non-existent value in the sum is been asked for. In that case the function will return *Nothing*.

```
instance (TypeEq l l′ b, Sum′ b l (HEither (l′, x) xs) x′)
    ⇒
    Sum l (HEither (l′, x) xs) x′
  where
    select l s = select′ b l s where b = typeEq l (⊥ :: l′)
    inject l x = inject′ b l x where b = typeEq l (⊥ :: l′)
```

The class *Sum* is instantiated via a helper class *Sum′*:

```
class Sum′ b l s x | b l s → x
  where
    select′ :: b → l → s → Maybe x
    inject′ :: b → l → x → s
instance Sum′ HTrue l (HEither (l, x) xs) x
  where
    select′ _ _ (HLeft (_, x)) = Just x
    inject′ _ l x = HLeft (l, x)
instance (Sum l ys y)
    ⇒
    Sum′ HFalse l (HEither lx ys) y
  where
    select′ b l (HRight ys) = select l ys
    inject′ b l y = HRight (inject l y)
```

Thus, the sole instance of *Sum* is constructed using the auxiliary class *Sum′* which has the same type of *Sum* plus an extra type-level boolean as argument. This boolean expresses whether the labels l and l' are the same, *i.e.* whether the injected or selected type is present at the left-most summand. This type level boolean is the result of the *typeEq* function from the *TypeEq* class, which is offered by the HList library for determining type-level equality.

4.2 Type of components

The first step to define a component's library amounts to providing a suitable type for whatever a component is. Following closely the coalgebraic model reviewed in Section 2, we arrive at

```
type CpTL s l i o m = s → l → i → m (s, (l, o))
```

Thus, the *CpTL* type constructor is synonymous to a function that receives a state s, a label l designating the operation to perform and its input i, and returns a pair with a new state and the label of the corresponding output o. The function result is suitably wrapped into a monad m which defines the behavioral model of the component.

Labels have an important role in component prototyping. Actually they act as 'port' identifiers for components and therefore define an interactive language which allows direct access to each operation's input or output port. A component language is an n-ary sum just like the one in the example above. But instead of defining a value of *HEither* type we resort to the *Encapsulate* class which *automatically* derives the component language for a given component.

```
class Encapsulate cp is st os m | cp → is st os m
  where
    (↠) :: cp → (st, is) → m (st, os)
```

As shown in the class definition, *cp* uniquely determines the remaining type parameters: the component input language – *is*, the state type – *st*, the output language – *os* and the behaviour monad – m.

The *Encapsulate* class has two instances. The first instance, shown below, deals with all the actions in the component but the last, while the second instance (omitted here because of its simplicity) takes care of the last element.

```
instance (Encapsulate (st → fs) is st os m, Monad m)
  ⇒
    Encapsulate (st → HCons (l′, e → m (st, r)) fs)
                (HEither (l′, e) is)
                st
                (HEither (l′, r) os)
                m
  where ...
```

The function definition of this instance is given in two parts: one applies when the label action is introduced in the language using a *HLeft* injection:

```
(↠) g (st, HLeft (_, e)) = do
                             let (HCons (l, f) _) = g st
                             (st′, r) ⟵ f e
                             return (st′, HLeft (l, r))
```

The other part of the function definition applies when a *HRgiht* injection is used:

```
(↠) fs (st, HRight is) = do
                           let (HCons _ fs′) = fs st
                           (st′, os) ⟵ (↠) (λst → fs′) (st, is)
                           return (st′, HRight os)
```

Having generated the component language, this is used to activate the corresponding prototype. This task is accomplished by another class:

```
class Apl it o1 l i st o m | l it o1 → i o
  where
    (@.) :: ((st, it) → m (st, o1)) → CpTL st l i o m
```

Class *Apl* provides both the input type i and the output type o of the component in each use of it.

```
instance (Monad m, Sum l o1 o, Sum l it i) ⇒
    Apl it o1 l i st o m
  where
    (@.) cp st l i = do
                       let input = inject l i :: it
                       (st′, output) ⟵ cp (st, input)
                       let (Just output′) = select l output
                       return (st′, (l, output′))
```

Note the use of the *inject* and *select* functions defined above and of the *Sum* class. The input to the component is generated by the injection of both the label and the input value.

The component is then activated with a state value and the input sum. The (monadic) output is selected again from a sum and returned together with the corresponding label.

4.3 Combinators

Now that a suitable encoding for component model has been defined, we proceed to describe the component combinators of the library.

4.3.1 *Machine activation – DoCompIO*

The first operator in the library is responsible for activating components as interactive prototypes[6].

```
class DoCompIO it o1 st o m
  where
    doCompIO :: ToIO m ⇒
      ((st, it) → m (st, o1)) → StateT st IO o
```

This class has a function which turns a (*CpTL*) component into an interactive state machine [15]. The machine will ask for an action and respective input (which is only possible if the language of the component is an instance of the Haskell standard *Read* class). Afterwards operator .@ (which applies the component to the state and the input) activates the component and the state machine evolves to the next state[7].

```
instance (Read it, Show o1)
    ⇒
    DoCompIO it o1 st o m
  where
    doCompIO cp = do
                      lift $ putStr "\nAction: "
                      i ⟵ lift getLine
                      st ⟵ get
                      let res = (.@) cp st (read i)
                      (st′, out) ⟵ lift $ toIO res
                      lift $ putStrLn $ show out
                      put st′
                      doCompIO cp
```

This interactive cycle will stop as soon as the machine receives a signal to die or whenever the read of the input fails. See Section 5 for an example of this operator in use.

4.3.2 *External choice –* ⊞

The *choice* operator allows the combination of two components: as the name indicates, it permits the activation of one component *or* the other, but never both at the same time. The language of the new component is the a sum of the languages of its arguments.

```
data Lft a = Lft a
data Rgt a = Rgt a
```

We use these two data types to distinguish the labels of the first component (*Lft*) from those of the second one (*Rgt*). Input and output types are also changed, becoming the sum of input or output types, respectively.

[6]where *ToIO m* turns the monad *m* into the monad *IO*.
[7]The $ :: $(a \to b) \to a \to b$ function just applies a function to its argument.

```
class Choice s1 l i o l1 s2 l′ i′ o′ l2 m cp3 |
          s1 l i o l1 s2 l′ i′ o′ l2 m → cp3
  where
    (⊞) :: (s1 → HCons (l, i → m (s1, o)) l1) →
         (s2 → HCons (l′, i′ → m (s2, o′)) l2) →
         cp3
```

This class declares that, given two components, a new one will be determined (*cp3*)[8].

```
instance (...)
    ⇒
    Choice s1 l i o l1 s2 l′ i′ o′ l2 m
         ((s1, s2) →
         (HCons (Lft l, Either i i′ → m ((s1, s2),
                        Either o o′)) lstf))
  where
    cp1 ⊞ cp2 =
      λ(s1, s2) → hAppend
                  (leftE (toLeftLst (cp1 s1) (s1, s2))
                        (⊥ :: i′) (⊥ :: o′))
                  (rightE (toRightLst (cp2 s2) (s1, s2))
                        (⊥ :: i) (⊥ :: o))
```

This instance constructs the new component with the help of another two classes, *ToLeftLst* and *ToRightLst*, which create the new language with the *Lft* and *Rgt* data types. The final step is to create the new input and output types (*Either i i′* and *Either o o′* respectively), which are a sum of the input and output types of the two given components (*i* and *i′*, and *o* and *o′* respectively). This is performed by the *leftE* and *rightE* which the reader can see in detail in the library source code.

4.3.3 *Parallel composition –* ⊠

In contrast to the choice combinator ⊞, parallel composition ensures that both composed components run at the same time. The language is composed of pairs: each pair has a label of the first component and a label of the second one, in this order.

```
class Parallel s1 l i o l1 s2 l′ i′ o′ l2 m cp3 |
           s1 l i o l1 s2 l′ i′ o′ l2 m → cp3
  where
    (⊠) :: (s1 → HCons (l, i → m (s1, o)) l1) →
         (s2 → HCons (l′, i′ → m (s2, o′)) l2) →
         cp3
```

The new language is constructed using the *ToPairs* class and its function *toPairs*, which receives the list of functions of both components and the two states and returns a new list of functions. Each function has as label a pair with a label from the first component and a label from the second one, in this order. Input and output types are now pairs with the input and output types of the two supplied components.

```
instance (...)
    ⇒
```

[8]Notice that components are split into state, language, input and output type and behavior monad.
Either a b = Left a | Right b represents datatype disjoint sum.

```
    Parallel s1 l i o l1 s2 l′ i′ o′ l2 m
              ((s1, s2) → HCons ((l, l′), (i, i′) →
                              m ((s1, s2), (o, o′))) lstf)
  where
    cp1 ⊠ cp2 =
      λ(s1, s2) → toPairs (cp1 s1) (cp2 s2) (s1, s2)
```

4.3.4 *Hook* – ↰

This operator allows to "feed back" a component with (part of) its own output. To implement this, a list with all the feed back rules must be created. This list has the following syntax:

```
(new_act_1, (old_act_11, old_act12)) .∗.
(new_act_2, (old_act_21, old_act12)) .∗.
...                                  .∗.
HNil
```

The first line above indicates that the result of the action *old_act_11* should be fed back as an input to the action *old_act_12*. *new_act_1* is the new action and could be different from *old_act_11*.

Given a component *cp* and a feed back list *l* as above, ↰ computes the new component cp'. The input and output type of the component must be framed as an Haskell *Either*.

```
class Hook cp l cp′ | cp l → cp′
  where
    (↰) :: cp → l → cp′
```

Suppose then the input type is *Either i z* and the output type *Either o z*. When the first execution succeeds, its output is tested. If it is a *Left i* then it is returned as result, but if it is a *Right z* then it is fed back to the component.

```
instance (...)
      ⇒
      Hook (s → lf) l (s → lf′)
  where
    (↰) f l = λs → let cons = constH f s l
                       lfr = hsnd HNil l
                       fs′ = deleteMany lfr (f s)
                   in cons ‘hAppend‘ fs′
```

This operator has four different stages to be performed. It starts by creating the new actions of the component (as described above) using the *constH* function. It will then infer the operations to be removed from the interface of the new component with the *hsnd* function. This list is then used to hide the operations (*deleteMany*). Finally, the new operations and the old ones that were not hidden are put together in the final list of operations.

4.3.5 *Refact*

This operator allows hiding and renaming of actions.

```
class Refact cp l cp′ | cp l → cp′
  where
    refact :: cp → l → cp′
```

The operator receives a component and a special list with the actions to hide and the renamings to be performed, splited into two different lists. The renamings are pairs relating the old action name, for example, *Lft* $ *Lft* $ *Rgt* $ *Lft reset* to the new name one, for example, just *new_reset*. This gives the possibility to simplify a lot the component's language as well as to block some actions in the interface. Such lists are specified according to the following syntax.

```
remove = act1 .∗. act2 .∗. ... .∗. HNil
redef =
  (old_act, new_act1 .∗. new_act2 .∗. ... .∗. HNil) .∗.
  (old_act, new_act1 .∗. new_act2 .∗. ... .∗. HNil) .∗.
  ...
  HNil
```

This allows for port replication through renaming an action to several different new identifiers.

```
instance (...)
      ⇒
      Refact (s → lf) (HCons l l′) (s → ftf)
  where
    refact f (HCons l l′) =
             λs → let fs = f s
                      ct = constl fs l′
                  in remov l (fs ‘hAppend‘ ct)
```

The *remov* function is responsible for hiding and *constl* for all the renamings. Because these are quite complex functions, they will not be shown here and the reader is referred to the library code.

4.3.6 *Wrap* – [ ]

As the name suggests, this operator wraps a component with input type i and output type o, given a function with type $i' \to i$ and a function with type $o \to o'$. This produces a component with input type i' and output type o'.

```
class Wrap cp f g cp′ | cp f g → cp′
  where
    wrap :: cp → f → g → cp′
```

In the only instance of this class the list of the component's ports is generated and passed to $wrap'$ together with the two wrapping functions.

```
instance (...)
      ⇒
      Wrap cp f g cp′
  where
    wrap cp f g = λs → wrap′ (cp s) f g
```

This auxiliary class returns the new list of input/output ports after wrapping.

```
class Wrap′ lf f g lf′ | lf f g → lf′
  where
    wrap′ :: lf → f → g → lf′
```

Besaides an instance to deal with the stop case (treats the empty list), another instance was created, in which every action of the component gets its input and output wrapped by the given wrapping functions.

```
instance (...)
    ⇒
    Wrap′ (HCons (l, i → m (s, o)) r) (i′ → i) (o → o′)
          (HCons (l, i′ → m (s, o′)) rt)
  where
    wrap′ (HCons (l, f1) r) f g =
         HCons (l, λi′ → do (s′, o′) ⟵ f1 (f i′)
                            return (s′, g o′))
               (wrap′ r f g)
```

4.3.7 *Lift* – ⌜⌝

The *Lift* operator creates a component from a function. A label must be supplied.

```
class Lift i o s l m cp | i o s l m → cp
  where
    cpLift :: (i → o) → l →
              (s → HCons (l, i → m (s, o)) HNil)
```

This label forms the language of the new created component, which has, of course, a single action.

```
instance Monad m
    ⇒
    Lift i o s l m (s → HCons (l, i → m (s, o)) HNil)
  where
    cpLift f l =
         λs → HCons (l, λi → return (s, f i)) HNil
```

This combinator allows the integration of existent functions in the a component based design.

4.3.8 *Pipeline* – ;

Sequential composition understood as a component pipeline mechanism is a fundamental pattern in a component-based programming style. Its implementation is given by the following class.

```
class Pipeline cp1 cp2 cp3 | cp1 cp2 → cp3
  where
    (;) :: cp1 → cp2 → cp3
```

The code listed bellow is the only instance of this class.

```
instance (...)
    ⇒
    Pipeline (s1 → lf1) (s2 → lf2) ((s1, s2) → lf)
  where
    cp1 ; cp2 =
       λ(s1, s2) → composeAll (cp1 s1) (cp2 s2)
```

This instance uses the *ComposeAll* class and its function *composeAll* to perform the composition of the functions.

5. REVISITING THE EXAMPLE

In this section we show in detail how the voting system presented in Section 2 can be prototyped using the component's type level library. As the reader will see, the implementation is quite straitforward, basically amounting to a direct translation of the theorical model.

5.1 Voting pad(s)

The voting system is constructed on top of two basis components: *vp* and *bb*. The implementation of *vp* is shown next:

```
vp = λn → (emit, emitf n) .∗. HNil
  where
    emitf n () = if n ≢ 0
                   then Just (n − 1, ())
                   else Nothing
```

where *emit* is defined as $emit = \bot :: Emit$. The *Emit* datatype has no constructors. This is the only label in the language of the *vp* component; function *emitf* is just the translation to HASKELL of the formal definition.

Suppose one wants a system with three voting pads. Within this model it is very easy to create a new component to represent this collection:

```
vp3 = vp ⊞ vp ⊞ vp
```

The input and output language of *vp3* is:

```
(HEither (Lft (Lft Emit),
          Either (Either () ()) ())
   (HEither (Lft (Rgt Emit),
             Either (Either () ()) ())
      (HEither (Rgt Emit,
                Either (Either () ()) ()) HVoid)))
```

5.2 Ballot box

Another piece of this voting system is the ballot box where the votes are counted and the system reset to a new value.

```
bb = λst → (reset, resetf st) .∗.
           (vote, votef st) .∗. HNil
  where
    resetf n (Left rv) = Just (rv, Left ())
    votef n (Right _) = Just (st − 1, Right (st − 1 ≡ 1))
```

This component has two actions: one to decrement the current state of the system (*vote*) and another to reset the system to a new value (*reset*). These two actions are defined similarly to *emit* above and constitute the language of this component. The input language is defined as follows

```
HEither (Reset, Either Int ())
        (HEither (Vote, Either Int ()) HVoid)
```

and the output is listed below.

```
HEither (Reset, Either () Bool)
        (HEither (Vote, Either () Bool) HVoid)
```

5.3 The voting system

Now that we have constructed the two main pieces of the system, we will plug them together to build the final voting system component. We start by composing the voting pad *vp3* with the co-diagonal *cod* as presented in Section 2. The *cod* component is a lift of the co-diagonal function ∇ given by:

$\textbf{data}\ \mathit{CodT}; \mathit{codT} = \bot :: \mathit{CodT}$
$\mathit{cod} = \mathit{cpLift} \triangledown \mathit{codT}$

Then the ballot box component is added to the model using combinator ⊞.

$\mathit{s3} = (\mathit{vp3}\ \ \mathbf{;}\ \ \mathit{cod})\ \boxplus\ \mathit{bb}$

This component has not the right type yet (has described in Section 2). It is necessary to apply the *wrap* operator to finally achieve the correct type and be able to use the ↰.

$\mathit{vs} = (\Lsh)\ (\mathit{wrap}\ \mathit{s3}\ a_{+}\ s_{+})\ \mathit{hp}$

hp is the list required by the ↰ operator to proceed with feed back.

$\mathit{hp} =$
$\quad (\mathit{emit1}, (\mathit{Lft}\ \$\ \mathit{Lft}\ \$\ \mathit{Lft}\ \$\ \mathit{Lft}\ \mathit{emit}, \mathit{vote}))\ .{*}.$
$\quad (\mathit{emit2}, (\mathit{Lft}\ \$\ \mathit{Lft}\ \$\ \mathit{Rgt}\ \$\ \mathit{Lft}\ \mathit{emit}, \mathit{vote}))\ .{*}.$
$\quad (\mathit{emit3}, (\mathit{Lft}\ \$\ \mathit{Rgt}\ \$\ \mathit{Lft}\ \mathit{emit}, \mathit{vote}))\ .{*}.\ \mathit{HNil}$

Each time a voting pad is activated two actions are performed: the first one is the local vote and the second one is the decrement of the state of the ballot box (*vote*). As the reader can notice, the labels for the voting pad activation were not used in the left side of the pair. We have created three new labels (*emit1*, *emit2* and *emit3*) to denote them and made them smaller to make easier its use. The final language for the input of the voting system is:

$\textbf{type}\ \mathit{Out} =$
$\quad \mathit{Either}\ (\mathit{Either}\ (\mathit{Either}\ (\mathit{Either}\ ()\ ())\ ())\ \mathit{Int})\ ()$
$\mathit{HEither}\ (\mathit{Emit1}, \mathit{Out})$
$\quad (\mathit{HEither}\ (\mathit{Emit2}, \mathit{Out})$
$\quad\quad (\mathit{HEither}\ (\mathit{Emit3}, \mathit{Out})$
$\quad\quad\quad (\mathit{HEither}\ (\mathit{Rgt}\ \mathit{Reset}, \mathit{Out})\ \mathit{HVoid})))$

and the output is shown below.

$\mathit{HEither}\ (\mathit{Emit1}, \mathit{Either}\ (\mathit{Either}\ ()\ \mathit{Bool})\ ())$
$\quad (\mathit{HEither}\ (\mathit{Emit2}, \mathit{Either}\ (\mathit{Either}\ ()\ \mathit{Bool})\ ())$
$\quad\quad (\mathit{HEither}\ (\mathit{Emit3}, \mathit{Either}\ (\mathit{Either}\ ()\ \mathit{Bool})\ ())$
$\quad\quad\quad (\mathit{HEither}\ (\mathit{Rgt}\ \mathit{Reset},$
$\quad\quad\quad\quad\quad \mathit{Either}\ (\mathit{Either}\ ()\ \mathit{Bool})\ ())\ \mathit{HVoid})))$

5.4 Animating the voting system

It is now possible to use the library to make component's prototype interactive. First it is necessary to create an instance of the standard *Read* class with the language of the component.

$\textbf{instance}\ (\mathit{Sum}\ \mathit{Emit1}\ s\ \mathit{In}, \mathit{Sum}\ \mathit{Emit2}\ s\ \mathit{In},$
$\quad\quad \mathit{Sum}\ \mathit{Emit3}\ s\ \mathit{In}, \mathit{Sum}\ (\mathit{Rgt}\ \mathit{Reset})\ s\ \mathit{In})$
$\quad \Rightarrow \mathit{Read}\ s$

For each action in the component's language it is necessary create a way of reading it. A possible encoding for *emit1* and *Rgt reset* reading is listed:

$\mathit{readsPrec}\ \_\ \texttt{"emit1"} =$
$\quad [(\mathit{inject}\ \mathit{emit1}\ (\mathit{Left}\ \$\ \mathit{Left}\ \$\ \mathit{Left}\ \$\ \mathit{Left}\ ()), \texttt{""})]$
$\mathit{readsPrec}\ \_\ (\texttt{'r'} : \texttt{'e'} : \texttt{'s'} : \texttt{'e'} : \texttt{'t'} : n) =$
$\quad [(\mathit{inject}\ (\mathit{Rgt}\ \mathit{reset})$
$\quad\quad\quad (\mathit{Left}\ \$\ \mathit{Right}\ (\mathit{read}\ n :: \mathit{Int})), \texttt{""})]$

The function which animates the voting system prototype is defined like this:

$\mathit{vsAnimation}\ () =$
$\quad \mathit{evalStateT}\ \$\ \mathit{doCompIO}\ ((\twoheadrightarrow)\ \$\ \mathit{vs}\ ())$

which provides for test interactive as illustrated below.

$\mathit{VotingSystem} > \mathit{vsAnimation}\ ()\ ((((20, 33), 14), ()), 4)$
$\mathit{Action} : \mathit{emit2}$
$(\mathit{Emit2}, \mathit{Left}\ (\mathit{Right}\ \mathit{False}))$
$\mathit{Action} : \mathit{emit1}$
$(\mathit{Emit1}, \mathit{Left}\ (\mathit{Right}\ \mathit{False}))$
$\mathit{Action} : \mathit{emit1}$
$(\mathit{Emit1}, \mathit{Left}\ (\mathit{Right}\ \mathit{True}))$
$\mathit{Action} : \mathit{reset}\ 3$
$(\mathit{Rgt}\ \mathit{Reset}, \mathit{Left}\ (\mathit{Left}\ ()))$
$\vdots$

6. CONCLUSIONS

This paper discussed the encoding of a formal model for state-based components into a concrete programming language. The theoretical framework is a theory of component *composition* in the sense that it lifts principles of classical modular construction [32] to the level of stand-alone, black-box software components. Actually, we start from abstracting a semantic model and, then, define a suitable *algebra*, *i.e.* a family of generic operators for assembling components together in a number of different ways. This calculus, which generalizes the algebra of Mealy machines, acts as a *glue code* for wiring autonomous components. Although its theory is detailed elsewhere (see *e.g.*, [5, 3] for the equational fragment and [26] for refinement issues), this paper shows how such combinators can be neatly and effectively implemented in HASKELL by exploring programming techniques at the type level.

This provides not only a smooth way to directly incorporate componentware in HASKELL, but also a testbed for prototyping software architectural patterns in a high-level programming language. Reference [6], for example, introduces a number of such patterns for composing *partial* components, which can be easily prototyped with the HASKELL library proposed here.

Related work

Note that commonly a component is regarded as a collection of objects and, therefore, component interaction is achieved by mechanisms implementing the usual *method call* semantics. Such perspective is typical of popular, wide-spread, technologies like, *e.g.*, CORBA[36], DCOM [12] or JAVABEANS [23].

An alternative point of view, inspired by research on coordination languages [11, 31], favors strict component decoupling in order to support a looser inter-component dependency. Here computation and coordination are clearly separated, communication becomes *anonymous* and component interconnection is externally controlled. This model is (partially) implemented in JAVASPACES on top of JINI [30] and fundamental to a number of approaches to componentware which identify communication by generic channels as the basic interaction mechanism — see, *e.g.*, REO [2] or PICCOLA [35, 28]. The focus becomes, therefore, the definition of external coordination devices which ensure the flow of data and enforce synchronization constraints within a component's network. A component's interface becomes, basically, a collection of *ports* through which values flow. The essential difference with respect to the approach adopted here lies in regarding software connectors as either component combinators (to produce new components from old) or as coordinators of data flow. In practice the expressive power of both approaches is comparable. Reference [7] includes a HASKELL implementation of a subset of a REO-inspired coordination model.

Related work includes Leijen *et al.* proposal to integrate COM [9] into HASKELL code [33, 22], which also provides both sequential and parallel composition mechanisms. Their starting point is, however, a concrete componentware platform whereas we are backed up by a full developed calculus which establishes a reasoning framework for analyzing and transform component based designs.

The technique of type-level programming was pioneered by McBride [24] and Hallgren [13] who explained how to the use HASKELL's type system as static logic programming language. Apart from heterogeneous collections [19], the technique has been used for lightweight dependently typed programming [24], implicit configurations [20], variable-length argument lists, formatting [14], and more.

Kiselyov *et al.* have developed a model of object-oriented programming inside Haskell [18], based on their HLIST library of extensible polymorphic records with first-class labels and subtyping. The model includes all conventional object-oriented features and more advanced ones, such as flexible multiple inheritance, implicitly polymorphic classes, and many flavors of subtyping. Silva *et al.* [37] used the same basis (HLIST records) and the same techniques (type-level programming) for modeling a different paradigm, *viz.* relational database programming. In the current paper we have added a third paradigm to the list: component-based development. All three models rely non-trivially on type-class bounded and parametric polymorphism.

Future work

Recently, integration of a (loose) notion of a HASKELL component within .NET has been addressed in different contexts (see. *e.g.*, [1]) to overcome the fact that typical HASKELL compilers do not provide support for XML-Web services, assisted GUI development, or HTML processing, which are frequent in most modern development frameworks. Other authors have tackled similar problems through specific *extensions* to HASKELL which provide primitives for concurrency [17], mobility [8] and distribution [10]. It would be an interesting issue for future work to study how the library proposed in this paper could be integrated to take advantage of such extensions.

In a wider perspective we would like to take the underlying theoretical framework and the library discussed here as a kernel, of a cross-platform environment for programming with reusable software components. Functional languages have been shown to lead to short development times and simplified maintenance for a wide range of applications. When integrated into a component model, such functional applications can be used for those parts of a project for which they are most suitable.

Availability

The HASKELL library of component combinators presented in this paper is available under the name HaMealy through the author's home pages (www.di.uminho.pt/˜jacome).

7. REFERENCES

[1] B. Alarcon and S. Lucas. Crossing the Rubicon: from Haskell to .NET through COM. *ERCIM News*, 63:51–52, October 2005.

[2] F. Arbab. Abstract Behaviour Types: a Foundation model for components and their composition. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 33–70. Springer Lect. Notes Comp. Sci. (2852), 2003.

[3] L. S. Barbosa. Towards a Calculus of State-based Software Components. *Journal of Universal Computer Science*, 9(8):891–909, August 2003.

[4] L. S. Barbosa. A Perspective on Component Refinement. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *FMCO'04, Third International Symposium on Formal Methods for Components and Objects – Revised Lectures*, pages 23–48. Springer Lect. Notes Comp. Sci. (3657), 2005.

[5] L. S. Barbosa and J. N. Oliveira. State-based Components Made Generic. In H. P. Gumm, editor, *CMCS'03, Elect. Notes in Theor. Comp. Sci.*, volume 82.1. Elsevier, 2003.

[6] L. S. Barbosa and J. N. Oliveira. Transposing Partial Components: an Exercise on Coalgebraic Refinement. *Theor. Comp. Sci.*, 365(1-2):2–22, 2006.

[7] M. A. Barbosa and L. S. Barbosa. A Relational Model for Component Interconnection. *Journal of Universal Computer Science*, 10(7):808–823, 2004.

[8] A. Bois, P. Trinder, and H. Loidl. mHaskell: Mobile computation in a purely functional language. *Journal of Universal Computer Science*, 11(7):1234–1254, 2005.

[9] K. Brockschmidt. *Inside OLE (2nd ed.)*. Microsoft Press, Redmond, WA, USA, 1995.

[10] F. H. Carvalho and R. D. Lins. Topological Skeletons in Haskell#. In *Proc. of IPDPS'03, International Parallel and Distributed Processing Symposium*. IEEE Press, 2003.

[11] D. Gelernter and N. Carrier. Coordination Languages and their significance. *Communication of the ACM*, 2(35):97–107, February 1992.

[12] R. Grimes. *Profissional DCOM Programming.* Wrox Press, 1997.

[13] T. Hallgren. Fun with Functional Dependencies. In *Proc. of the Joint CS/CE Winter Meeting*, pages 135–145, 2001. Dep.t of Computing Science, Chalmers, Göteborg, Sweden.

[14] R. Hinze. Formatting: a class act. *J. Funct. Program.*, 13(5):935–944, 2003.

[15] M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, pages 97–136, 1995.

[16] S. L. P. Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.

[17] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[18] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft of 10 September 2005, 2005.

[19] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004.

[20] O. Kiselyov and C. Shan. Functional pearl: implicit configurations–or, type classes reflect the values of types. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 33–44, New York, NY, USA, 2004. ACM Press.

[21] A. Kock. Strong Functors and Monoidal Monads. *Archiv für Mathematik*, 23:113–120, 1972.

[22] D. Leijen, E. Meijer, and J. Hook. Haskell as an Automated Controller. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Third International Summer School on Advanced Functional Programming, Braga*, pages 268–289. Springer Lect. Notes Comp. Sci. (1608), September 1998.

[23] V. Matena and B. Stearns. *Applying Entreprise JavaBeans: Component-Based Development for the J2EE Platform.* Addison-Wesley, 2000.

[24] C. McBride. Faking it – Simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, 2002.

[25] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell Systems Techn. Jour.*, 34(5):1045–1079, 1955.

[26] S. Meng and L. S. Barbosa. Components as coalgebras: The refinement dimension. *Theor. Comp. Sci.*, 351:276–294, 2005.

[27] R. Milner. *Communication and Concurrency.* Series in Computer Science. Prentice-Hall International, 1989.

[28] O. Nierstrasz and F. Achermann. A Calculus for Modeling Software Components. In F. S. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. First International Symposium on Formal Methods for Components and Objects (FMCO'02)*, pages 339–360. Springer Lect. Notes Comp. Sci. (2852), 2003.

[29] O. Nierstrasz and L. Dami. Component-oriented software technology. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice-Hall International, 1995.

[30] S. Oaks and H. Wong. *Jini in a Nutshell.* O'Reilly and Associates, 2000.

[31] G. Papadopoulos and F. Arbab. Coordination Models and Languages. In *Advances in Computers — The Engineering of Large Systems*, volume 46, pages 329–400. Centrum voor Wiskunde en Informatica (CWI), 1998.

[32] D. Parnas. Information Distribution Aspects of Design Methodology. In *Information Processing '72*, pages 339–344. North-Holland, 1972.

[33] S. Peyton Jones, E. Meijer, and D. Leijen. Scripting COM components from Haskell. In *Fifth International Conference on Software Reuse (ICSR'98)*, Victoria, B.C., Canada, Junho 1998. IEEE Computer Society Press.

[34] J. Rutten. Universal coalgebra: A theory of systems. *Theor. Comp. Sci.*, 249(1):3–80, 2000. (Revised version of CWI Techn. Rep. CS-R9652, 1996).

[35] J.-G. Schneider and O. Nierstrasz. Components, Scripts, Glue. In L. Barroca, J. Hall, and P. Hall, editors, *Software Architectures - Advances and Applications*, pages 13–25. Springer-Verlag, 1999.

[36] R. Siegel. *CORBA: Fundamentals and Programming.* John Wiley & Sons Inc, 1997.

[37] A. Silva and J. Visser. Strong types for relational databases. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 25–36, New York, NY, USA, 2006. ACM Press.

[38] C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[39] P. Wadler and K. Weihe. Component-Based Programming Under Different Paradigms. Technical report, Dagstuhl Seminar 99081, February 1999.