Protocol Specialization

Matthias Neubauer and Peter Thiemann

Institut fär Informatik, Universität Freiburg, Georges-Köhler-Allee 079,
79110 Freiburg, Germany
{neubauer,thiemann}@informatik.uni-freiburg.de

Abstract. In component-based programming, the programmer assembles applications from prefabricated components. The assembly process has two main steps: adapting a component by tweaking its configuration parameters, and connecting components by gluing output interfaces to input interfaces. While convenient, this approach may give rise to code bloat and inefficiency because prefabricated code is overly general, by necessity.
The present work addresses ways to remove unnecessary code during the deployment of a closed set of components by using program specialization. Our framework models components at the intermediate language level as systems of concurrent functional processes which communicate via channels. Each channel acts as a component connector with the interface determined by the channel's protocol. We present an analysis that determines the minimum protocol required for each process and specify the specialization of a process with respect to a desired protocol, thereby removing unnecessary code.
The resulting specialization algorithm is unique in that it processes a concurrent base language, terminates always, and is guaranteed not to expand the program beyond its original size.

1 Introduction

Component-based programming [4] changes the programmer's focus of attention from low-level algorithmic issues to high-level concerns of assembling components with matching interfaces. Components are designed for generality so that they are reusable in a number of different situations. Hence, they provide ways of adapting them to the special needs of the application, *e.g.*, by setting configuration parameters. As is often the case, generality and programming convenience come at the price of increased demands on resources like execution time and memory. Regaining this lost efficiency requires specialization of the assembled application.

Partial evaluation [15] is a successful program specialization technique that has been applied to component adaption in the past [22, 1]. However, specialization becomes more complicated in the presence of concurrency, if components replace or enhance procedural interfaces by events and communication—as customary in component frameworks for graphical user interfaces—or if resource

constraints must be observed. For example, partial evaluation often leads to uncontrolled code growth and the specialization of concurrent programs has not received much attention, yet.

The goal of the present investigation is to identify a specialization methodology that achieves meaningful results under resource constraints and in the presence of communication-based interfaces. The particular scenario that we investigate is one where sequential components that run concurrently communicate with each others through bidirectional channels. Some components act as servers in that they only react to queries from other components and others act as clients. An instance of such a scenario is a wireless intranet where a company provides an internal messaging service to a range of different handheld computers. Each brand of handheld comes with a component implementing the service protocol possibly with vendor-specific extensions. Since space is at premium in a handheld, this component should be as lean as possible. Our framework provides a means of specializing the handheld components with respect to the protocol actually employed by the servers. Our specializer removes the vendor-specific extensions as well as any functionality not referred to by the server. Furthermore and unlike other specializers, our specializer is guaranteed to shrink the program so that it consumes less space.

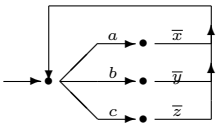
Our investigation focuses on channel-based communication between components because it subsumes traditional procedure-based interfaces. We assume that the entire system is described by a program in Gay and Hole's concurrent functional language with session types [9] with each component modeled by one process.

The present work makes the following contributions. We simplify and streamline Gay and Hole's language to a low-level concurrent functional language. Next, we extend the language's type system with subtyping and singleton types. In Section 5, we specify a notion of *slice types* to formally talk about specialization opportunities on the type level. Subtyping and singleton types are essential for this step. Finally, we put the pieces together and specify the specialization algorithm. Compared to other published specializers, our algorithm has three unique features. First, it deals with a concurrent base language. Second, it is guaranteed to terminate on all inputs, and third, it guarantees that the specialized program is no larger than the original one.

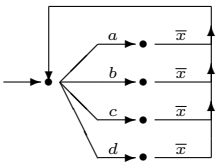
The type system underlying our work relies on recursive types with singleton types and subtyping. The combination with singleton types is a novel contribution to session type systems. Linear typing is also involved for tracking the state of a communication channel.

2 An Example Protocol Specialization

From a bird's eye view, the task of protocol specialization looks simple. Just weed out the unnecessary parts and you are done! Unfortunately, identifying the unnecessary parts is not trivial and care must be taken not to increase the size of the original program in the specialization process.

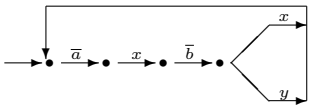


Server A

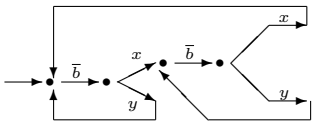


Server B

Fig. 1. Server components



Client C



Client D

Fig. 2. Client components

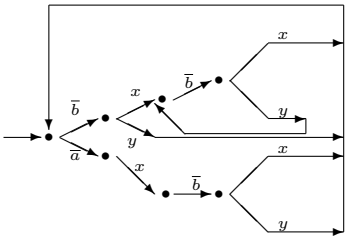
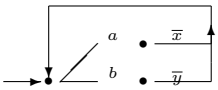
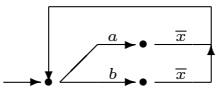


Fig. 3. Greatest common channel type



Server A



Server B

Fig. 4. Slice types of the servers

Recall the messaging service scenario from the introduction with two different servers providing the same service, replicated for robustness and fault tolerance. Each server comes from a different vendor, thus each of them has vendor-specific enhancements beyond the required base protocol. For simplicity, we assume that client processes attach nondeterministically to exactly one server process. If all servers are connected, then additional clients must wait for a free server. Servers that service more than one client can be modeled in our calculus but would lead to on overly complicated example.

Figure 1 contains a description of the external communication behavior[1] of the two servers written as a finite automaton and Figure 5 contains sample code for server A. Each state corresponds to some internal computation at the peer and each transition corresponds to a communication operation. The annotation $a, \ldots$ on a transition denotes the ability to receive inbound messages, whereas the annotation $\overline{x}, \ldots$ denotes sent messages. Each path through an automaton corresponds to a possible behavior.

For example, after some initial computation, server A waits for one of the inputs a, b, or c. After receiving one of these inputs, A performs some computations and outputs $\overline{x}$, $\overline{y}$, or $\overline{z}$, respectively, then it may perform some more internal

[1] Essentially, a graphical rendition of the channel types that will be introduced below.

```
Let p = NewPort in

Let Rec serverA (n) =
    Let c = Listen (p) in
    Let Rec serv [c] (n) =
        Receive (c)
          [a (m): Let n1 = Opsa (n, m) in
                  Let Send c(x (n1)) in
                  serv [c](n1)
          ,b (m): Let n1 = Opsb (n, m) in
                  Let Send c(y (n1)) in
                  serv [c](n1)
          ,c ():  Let Send c(z (n1)) in
                  serv [c](n1)
          ]
    in serv [c](n)
in
```

```
Let Rec clientC (n) =
    Let c = Connect (p) in
    Let Rec clie[c] (n) =
      Let Send c(a (n)) in
      Receive (c)
        [x (m):
          Let m1 = Opc (n, m) in
          Let Send c (b (m1)) in
          Receive (c)
            [x (m): Let m2 = Opcx (n, m) in
                    clie[c] (m2)
            ,y (m): Let m2 = Opcy (n, m) in
                    clie[c] (m2)
            ]
        ]
    in clie[c] (n)
in

serverA (0) || clientC (0)
```

Fig. 5. Code Sample for Server A and Client C Before Specialization

```
...
Let Rec serverA (n) =
    Let c = Listen (p) in
    Let Rec serv [c] (n) =
        Receive (c)
          [ a (m): Let n1 = Opsa (n, m) in Let Send c(x (n1)) in serv [c](n1)
          , b (m): Let n1 = Opsb (n, m) in Let Send c(y (n1)) in serv [c](n1) ]
    in serv [c](n)
in
...
```

Fig. 6. Server A after specialization

computation before it waits again for one of the inputs. The calculus in Section 3 codifies this behavior with the channel type $\mu\beta.[a : [\overline{x} : \beta], b : [\overline{y} : \beta], c : [\overline{z} : \beta]]$, which is slightly harder to read than its automata rendition in Figure 1.

Figure 2 describes the two client components (see Figure 5 for code of client C) corresponding to different brands of handheld computers. Either client is capable of communicating to either server. This may be seen by tracing each possible pair of client and server.

Assuming that A, B, C and D are the only kinds of peers involved in the intranet, it is clear that clients C and D can only ever connect to servers A and B and, vice versa, clients C and D are the only clients of servers A and B.

Since the structure of the clients is quite different, it does not make sense to tailor a server to one of the clients. Also, it would not make sense to make a fixed assignment of one server to a particular client because this assignment would defeat the robustness intended by the designer of the architecture. Hence, the first task is to find a smallest common description that subsumes the behavior of both clients and tailor both servers to this common description.

A common description of the clients' view of the channel can be constructed graphically by overlaying the two descriptions as shown in Figure 3. The general construction is more involved because client descriptions may have different loop structures that must be synchronized for constructing the common descriptions.

The common client type forms the basis for specializing both servers. However, specializing a server directly from the common client type may lead to code duplication. For example, the processing of the message b would be replicated two times in the specialized server. For that reason, we use the common client type to compute for each server a slice type which specializes the server type towards the client type while still retaining the structure of the original server type. Since the structure of the original server type closely mirrors the structure of its implementation, we can guarantee that no code is duplicated. Figure 4 contains the slice types computed for the two servers, and Figure 6 shows the specialized code for server A.

In the example, the client components cannot be specialized, but in the general case, there might be specialization opportunities in the clients, too. The scenario is dual with the roles of clients and servers reversed. The required steps are analogous: find a common type for the servers, compute an individual slice type for each client, and then specialize the client with respect to the slice type.

Interestingly, the same framework also handles adaption with respect to configuration parameters. To see this, we have to assume that configuration parameters are presented to components via communication channels. In this scenario, each component acts as a client for its configuration channel(s) and the configuration parameters are supplied through a special configuration component. To take advantage of the actual values on these channels requires introducing singleton types into the type system. The specializer can then decide conditionals and perform primitive operations based on the additional knowledge. Specialization still does not expand the program as no unfolding of function calls takes place.

The subsequent sections first introduce a formal calculus for expressing client and server components. Then, each of the steps outlined above is put on a formal basis and it is proven that the specialization algorithm is well-defined. The soundness of the specialization follows from the soundness of subtyping in the calculus.

3 Calculus with Channel Types

This section defines the λ_{CS} calculus, a formal calculus for expressing client and server components. After presenting the syntax, a type system fixes the static semantics of λ_{CS}.

3.1 Syntax

The calculus deals with five kinds of data, first-order base type values, port values, channel values, functions, and labels. Labels have a status similar to labels in record and variant types. They occur in channel types and they can be sent and received via channels: each message consists of a value tagged with a label. Channel values must be treated linearly: during execution each channel end must have exactly one reference. Each operation on a channel changes its

state and the close operation destroys the channel. In our version of the calculus channels may neither be sent nor received over a channel themselves.[2]

Labels	l	$\in$ Label
Variables	x	$\in$ Var
Channel Variables	c	$\in$ ChannelVar
Definitions	d	$::= x = i \mid x = \texttt{Op}(\tilde{x}) \mid \texttt{rec}\ x[\tilde{c}](\tilde{x}) = e \mid x = \texttt{NewPort}$
		$\mid\ c = \texttt{Connect}(x) \mid c = \texttt{Listen}(x) \mid \texttt{Send}\ c(\bar{l}\ x) \mid \texttt{Close}\ (c)$
Expressions	e	$::= \texttt{Halt} \mid \texttt{Let}\ d\ \texttt{in}\ e \mid \texttt{If}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid x\ [\tilde{c}]\ (\tilde{x})$
		$\mid\ \texttt{Receive}\ c[g] \mid e \parallel e$
	g	$::= l(x) \to e \mid g, g$

The expressions of the calculus come in a sequentialized style reminiscent of continuation-passing style. That is, an expression e is a sequence of definitions which ends in either a `Halt` instruction, a conditional, a function call, a receive instruction that branches on the received label, or a concurrent execution of two expressions. All argument positions are restricted to variables. Without loss of generality, we assume that the set of labels is totally ordered, and branches of a receive instruction always occur ordered with respect to labels. The notation $\tilde{x}$ stands for $x_1, \ldots, x_n$ where n is determined by the context. Analogously, $(\tilde{x} : \tilde{\tau})$ stands for $(x_1 : \tau_1) \ldots (x_n : \tau_n)$.

A definition d is either the creation of a constant, the application of a primitive operation, the definition of a recursive function, the send operation, the creation of a new communication port, or an administrative operation on a channel: closing the channel, connecting to a channel (client connection), and listening to a channel (server connection).

3.2 Static Semantics

The following section introduces the type language for our calculus including a type language for channels, called channel types, and presents the type system.

Types	τ	$::= b \mid [\tilde{\gamma}]\tilde{\tau} \to 0$
Channel types	γ	$::= \varepsilon \mid [\eta] \mid \beta \mid \mu\beta.\gamma$
	η	$::= \ell(\tau) : \gamma \mid \eta, \eta$
	ℓ	$::= \bar{l} \mid l$
Port types	ζ	$::= \texttt{Port}\ \gamma$
Type Environments	Γ	$::= \emptyset \mid \Gamma(x : \tau)$
Channel Environments	Θ	$::= \emptyset \mid \Theta(c : \gamma)$

A type is either a base type, a function type, or a port type. Due to the sequential style, functions do not return values. Instead they must take a continuation argument. Function arguments are split in two lists, one for linear arguments carrying channel values and one for other arguments.

A channel type is either empty (the channel is closed), the empty word (the channel is depleted but not yet closed), a label-tagged alternative of different

[2] Extending the calculus in this way is not hard, but not essential for the present work.

channel types (the value may be sent $\bar{l}(\tau)$ or received $l(\tau)$), or a type variable which is used in constructing a recursive type with the μ operator. The μ operator constructs a fixpoint, *e.g.*, $\mu\beta.\gamma \approx \gamma[\beta \mapsto \mu\beta.\gamma]$. All uses of μ are *expansive*, that is, there can be no subterms of the form $\mu\beta_1 \ldots \mu\beta_n.\beta_1$.

The type system relies on two judgments, $\Theta, \Gamma \vdash e$, to check the consistency of an expression with respect to channel environment Θ and type environment Γ and $\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma'$ to model the effect of a definition as a transformation of the environments. The rules are shown in Figures 7 and 8 respectively. For now, $\leq$ can be read as syntactic equality. Section 4 defines a suitable subtyping relation.

The `Halt` rule requires that all channels are closed. The conditional passes the environments unchanged to both branches. The let expression types the body after transforming the environment according to the definition. Applying a function requires that the function consumes all remaining channels. Receiving a tagged value eliminates a labeled alternative in the channel's type. The branches are checked with the channel type updated according to the alternative taken. For concurrent execution, the channel environment is split in two disjoint parts whereas the type environment is passed to both subexpressions.

A primitive operation has arguments and result of base type. It does not depend on the channel environment. Creating a new port guesses a port type and attaches it to a port value. Ports serve as mediators through which clients and servers can agree on a common channel type. Sending of a labeled value selects the appropriate component of the channel type for the rest of the sender expression and changes the channel environment accordingly. Function formation is independent of the current channel environment. The body of the function must be checked with the channel environment prescribed by the channels passed at the call site of the function. Closing a communication channel requires that there are no exchanges left in its type. `Listen` creates a new server channel attached to a particular port. `Connect` creates a new client channel for a port. Since the channel type registered with the port describes the communication behavior of the server, the client processes the channel using the *mirrored* type, $\overline{\gamma}$, with inbound and outbound labels exchanged. Mirrored types are formally defined in Figure 12. Note that mirroring a type does not affect messages.

4 Specialization Opportunities

Specialization frameworks often include a notion of binding-time analysis [15]. Such an analysis determines which values are available to specialization without actually running the program or fixing a particular value. Similar, but less permissive information can be obtained by introducing subtyping and singleton types to λ_{CS}'s type system.

$$\emptyset, \Gamma \vdash \texttt{Halt} \quad \frac{\Gamma(x) \leq b \qquad \Theta, \Gamma \vdash e_1 \qquad \Theta, \Gamma \vdash e_2}{\Theta, \Gamma \vdash \texttt{If } x \texttt{ then } e_1 \texttt{ else } e_2}$$

$$\frac{\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma' \qquad \Theta', \Gamma' \vdash e}{\Theta, \Gamma \vdash \texttt{Let } d \texttt{ in } e} \quad \frac{\Gamma(x) = [\tilde{\gamma}]\tilde{\tau} \to 0 \qquad \Gamma(\tilde{z}) \leq \tilde{\tau} \qquad \Theta \leq (\tilde{c} : \tilde{\gamma})}{\Theta, \Gamma \vdash x\ [\tilde{c}]\ \tilde{z}}$$

$$\frac{\Theta(c) \leq [l_i(\tau_i) : \gamma_i]_{i=1}^n \qquad (\forall i)\ \Theta(c : \gamma_i), \Gamma(x_i : \tau_i) \vdash e_i}{\Theta, \Gamma \vdash \texttt{Receive } c[l_i(x_i) \to e_i]_{i=1}^n} \quad \frac{\Theta_1, \Gamma \vdash e_1 \qquad \Theta_2, \Gamma \vdash e_2}{\Theta_1 + \Theta_2, \Gamma \vdash e_1 \parallel e_2}$$

Fig. 7. Typing rules for expressions

$$\Theta, \Gamma \vdash x = i \Rightarrow \Theta, \Gamma(x : b) \quad \Theta, \Gamma \vdash x = \texttt{NewPort} \Rightarrow \Theta, \Gamma(x : \texttt{Port } \gamma)$$

$$\frac{\Gamma(x_i) \leq b}{\Theta, \Gamma \vdash x = \texttt{Op}(x_1, \ldots, x_n) \Rightarrow \Theta, \Gamma(x : b)} \quad \frac{\Gamma(x) \geq \texttt{Port } \gamma}{\Theta, \Gamma \vdash \texttt{Listen}(x) \Rightarrow \Theta(c : \gamma), \Gamma}$$

$$\frac{(\tilde{c} : \tilde{\gamma}), \Gamma(f : [\tilde{\gamma}]\tilde{\tau} \to 0)(\tilde{x} : \tilde{\tau}) \vdash e}{\Theta, \Gamma \vdash \texttt{rec } f[\tilde{c}](\tilde{x}) = e \Rightarrow \Theta, \Gamma(f : [\tilde{\gamma}]\tilde{\tau} \to 0)} \quad \frac{\Gamma(x) = \texttt{Port } \gamma' \qquad \gamma' \leq \overline{\gamma}}{\Theta, \Gamma \vdash c = \texttt{Connect}(x) \Rightarrow \Theta(c : \gamma), \Gamma}$$

$$\frac{\Gamma(x_j) \leq \tau_j \qquad \gamma = [\overline{l_i}(\tau_i) : \gamma_i]_{i=1}^n}{\Theta(c : \gamma), \Gamma \vdash \texttt{Send } c(\overline{l_j}\ x_j) \Rightarrow \Theta(c : \gamma_j), \Gamma} 1 \leq j \leq n \quad \Theta(c : \varepsilon), \Gamma \vdash \texttt{Close } (c) \Rightarrow \Theta, \Gamma$$

Fig. 8. Typing rules for definitions

$$\frac{\Gamma(x) = \mathbf{S}\{i\ :\ b\} \quad i \neq 0 \qquad \Theta, \Gamma \vdash e_1}{\Theta, \Gamma \vdash \texttt{If } x \texttt{ then } e_1 \texttt{ else } e_2} \quad \frac{\Gamma(x) = \mathbf{S}\{0\ :\ b\} \qquad \Theta, \Gamma \vdash e_2}{\Theta, \Gamma \vdash \texttt{If } x \texttt{ then } e_1 \texttt{ else } e_2}$$

Fig. 9. Typing rules for expressions using Singleton Types

$$\Theta, \Gamma \vdash x = i \Rightarrow \Theta, \Gamma(x : \mathbf{S}\{i\ :\ b\})$$

$$\frac{(\forall j)\ \Gamma(x_j) = \mathbf{S}\{i_j\ :\ b\} \qquad \texttt{Op}(i_1, \ldots, i_n) \to i_0}{\Theta, \Gamma \vdash x = \texttt{Op}(x_1, \ldots, x_n) \Rightarrow \Theta, \Gamma(x : \mathbf{S}\{i_0\ :\ b\})}$$

Fig. 10. Typing rules for definitions introducing Singleton Types

$$(\textit{sub-s-s})\frac{}{\mathbf{S}\{i\ :\ b\} \leq \mathbf{S}\{i\ :\ b\}} \quad (\textit{sub-s-b})\frac{}{\mathbf{S}\{i\ :\ b\} \leq b} \quad (\textit{sub-b-b})\frac{}{b \leq b}$$

$$(\textit{sub-}\tau\textit{-arrow})\frac{(\forall i)\ \gamma_i' \leq \gamma_i \quad (\forall j)\ \tau_j' \leq \tau_j}{[\tilde{\gamma_i}]\tilde{\tau_j} \to 0 \leq [\tilde{\gamma_i'}]\tilde{\tau_j'} \to 0} \quad (\textit{sub-}\zeta)\frac{\gamma \leq \gamma'}{\texttt{Port } \gamma \leq \texttt{Port } \gamma'}$$

$$(\textit{sub-send})\frac{(\forall 1 \leq i \leq n)\ \tau_i' \leq \tau_i \quad \gamma_i \leq \gamma_i'}{[\overline{l_i}(\tau_i) : \gamma_i]_{i=1}^{n+k} \leq [\overline{l_i}(\tau_i') : \gamma_i']_{i=1}^n} \quad (\textit{sub-recv})\frac{(\forall 1 \leq i \leq n)\ \tau_i \leq \tau_i' \quad \gamma_i \leq \gamma_i'}{[l_i(\tau_i) : \gamma_i]_{i=1}^n \leq [l_i(\tau_i') : \gamma_i']_{i=1}^{n+k}}$$

$$(\textit{sub-}\mu\textit{-fold-right})\frac{\gamma \leq \gamma'[\beta \mapsto \mu\beta.\gamma']}{\gamma \leq \mu\beta.\gamma'} \quad (\textit{sub-}\mu\textit{-fold-left})\frac{\gamma[\beta \mapsto \mu\beta.\gamma] \leq \gamma'}{\mu\beta.\gamma \leq \gamma'}\gamma' \neq \mu\beta.\gamma''$$

Fig. 11. Subtyping

$$\overline{\varepsilon} = \varepsilon \qquad \overline{\beta} = \beta \qquad \overline{\eta, \eta} = \overline{\eta}, \overline{\eta} \qquad \overline{\overline{l}} = l$$
$$\overline{[\eta]} = [\overline{\eta}] \qquad \overline{\mu\beta.\gamma} = \mu\beta.\overline{\gamma} \qquad \overline{\ell(\tau) : \gamma} = \overline{\ell}(\tau) : \overline{\gamma} \qquad \overline{l} = \overline{l}$$

Fig. 12. Mirroring of Types

4.1 Singleton Types

Our notion of singleton types is restricted to base types. Hence, we extend the type language as follows.

$$\tau ::= \ldots \mid \mathbf{S}\{i : b\}$$

The type $\mathbf{S}\{i : b\}$ represents a particular base value, i, at the type level. Singleton types may occur wherever ordinary types are expected, also in channel types. Figures 9 and 10 show refinements of the existing typing rules facilitates by exploiting singleton types where possible. Namely, declarations of constant bindings and declarations of operator applications of statically known base values introduce singleton types. A conditional can be typed less strictly if the condition has a singleton type.

4.2 Subtyping

Following Gapeyev et al. [8], the subtype relation is the greatest fixed point of a generating function. We specify the generating function by means of inference rules shown in Figure 11. The resulting subtyping relation gives rise to a more precise instance of the typing rules in the previous section. Subtyping derives from two sources: singleton types and label-tagged alternatives occurring in channel types [20, 10].

Because a value of channel type with a sending capability as its first component can be substituted by other channels that allow "smaller" values to be transmitted instead, subtyping of outbound channel types must be contravariant. On the other hand, a channel with receiving capability can always be demoted to a type that just receives "larger" types, so subtyping for incoming channel types must be covariant.

The rule for function types extends the subtyping relation to functions in the usual way. The additional rules involving recursive μ-types explain the behavior of right and left μ-folding with respect to subtyping. The folding rules are asymmetric to makes them non-overlapping and thus invertible. Subtyping extends to port types covariantly.

5 Protocol Specialization

We divide the framework for protocol specialization into two parts. In the first subsection, we characterize slice types as potential target types for specialization. The second subsection specifies specialization with respect to a slice type.

5.1 Slice Types

As explained in the introductory section, our goal is to specialize code of a component in such a way that fragments of the code that are never used while communicating with other components of the system are removed. The relation

between the actual behavior of a component, the behavior expected by all its communication partners, and the designated behavior without unreachable code is formally stated as a relation on types. Given two channel types for a channel, the one corresponding to the communication pattern of the actual implementation, and a supertype specifying the expected behavior of all communication partners, we are seeking for an intermediate channel type which only includes required communication branches and which also exhibits the same recursive structure as the channel type corresponding to the actual source code. We call intermediate channel types with such properties *slice types*.

We again define generating functions for two relations specifying lower and upper slice types by means of inference rules. In turn, the two relations are the greatest fixed points of the presented generating functions.

The relation $\gamma \trianglelefteq \gamma_s \leq \gamma'$ specifies the lower slice type relation between three channel types. It expresses two distinct features about the relation of the three types: first, all three channel types are related by the subtype relation, and second, the first channel type, γ, and the lower slice type, γ_s, exhibit the same recursive structure. The relation $\gamma \leq \gamma_s \trianglelefteq \gamma'$, specifies upper slice types. It is dual to the notion of a lower slice type by demanding that the slice type must have the same recursive structure as the upper channel type.

There are two sets of rules each defining possible lower slice types of two label-tagged alternatives, four rules for inbound channels and four rules for outbound channels, respectively. In the following, we exemplarily show the rules concerning outbound channels.

$$(\textit{l-slice-}\delta\textit{-send-1})\frac{\delta \leq \delta_s}{\delta \trianglelefteq \delta_s \leq []}$$

$$(\textit{l-slice-}\delta\textit{-send-2})\frac{l_i < l_k \leq l_j \quad \delta \trianglelefteq [\overline{l_k}(\tau^s) : \gamma^s, \delta^s] \leq [\overline{l_j}(\tau') : \gamma', \delta']}{[\overline{l_i}(\tau) : \gamma, \delta] \trianglelefteq [\overline{l_k}(\tau^s) : \gamma^s, \delta^s] \leq [\overline{l_j}(\tau') : \gamma', \delta']}$$

$$(\textit{l-slice-}\delta\textit{-send-3})\frac{l_i < l_j \quad \tau^s \leq \tau \quad \gamma \leq \gamma^s \quad \delta \trianglelefteq \delta^s \leq [\overline{l_j}(\tau') : \gamma', \delta']}{[\overline{l_i}(\tau) : \gamma, \delta] \trianglelefteq [\overline{l_i}(\tau^s) : \gamma^s, \delta^s] \leq [\overline{l_j}(\tau') : \gamma', \delta']}$$

$$(\textit{l-slice-}\delta\textit{-send-4})\frac{\tau' \leq \tau^s \trianglelefteq \tau \quad \gamma \trianglelefteq \gamma^s \leq \gamma' \quad \delta \trianglelefteq \delta^s \leq \delta'}{[\overline{l_i}(\tau) : \gamma, \delta] \trianglelefteq [\overline{l_i}(\tau^s) : \gamma^s, \delta^s] \leq [\overline{l_i}(\tau') : \gamma', \delta']}$$

To specify these rules concisely, we assume a total ordering $\leq$ on labels and also that labels are always listed in that order. Depending on the labels of the first branch of the lower and upper types, there are different possibilities to choose a first branch of a possible slice type. The subtype of an outbound alternatives may have additional alternative branches and branches starting with the same label impose a contravariant subtype relation on their transmitted values. The four rules handle different cases where the upper type does not have a branch at all, the lower and upper types start with different labels and the slice type label either corresponds to the lower label or not, or all three alternatives start with the same label.

The subtype of an inbound channel must have fewer alternative branches and the types of the incoming values behave covariantly. The rules relating singleton

types and base types, function types, and port types just carry over regular subtyping to three channel types, additionally taking into account the correct alignment of the recursive structure of the slice types. The folding rules for μ-types show that a lower slice type must exactly follow the folding of the lower channel type, γ. On the other hand, folding the upper channel type γ' must not affect the slice type at all, it stays unchanged while folding the lower type and the slice type. The whole set of rules is not shown due to lack of space but may be found elsewhere [18].

The rules specifying upper slice types are analogous to the rules specifying lower slice types. The only difference is that the folding of recursive slice types—that is, slice types in the shape of μ-types—is now synchronized with the folding of the upper channel type.

5.2 Translation

Instead of stating a specific implementation for our specialization scheme, we formulate valid specialization algorithms as instances of a type-based translation relation inspired by Hughes's type specialization [14].

To this end, we specify two relations expressing valid specializations of typed expressions and definitions. The translation relation for expressions $\Theta, \Gamma \vdash e \hookrightarrow \Theta^*, \Gamma^* \vdash e^*$, specifies that the typing $\Theta^*, \Gamma^* \vdash e^*$ for a new expression e^* is a specialized version of an original expression e with $\Gamma, \Theta \vdash e$.

Only two kinds of expression may change during specialization. First, the conditional simplifies to one of the branches if the type of the condition is a singleton type. The following rule exemplarily shows the specification of valid translations for conditionals where the conditional value is known to be zero—that is, the else branch can be chosen statically.

$$\frac{\Gamma(x) = \mathbf{S}\{0 : b\} \quad \Gamma^*(x) = \mathbf{S}\{0 : b\} \quad \Theta, \Gamma \vdash e_2 \hookrightarrow \Theta^*, \Gamma^* \vdash e_2^*}{\Theta, \Gamma \vdash \texttt{If}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \hookrightarrow \Theta^*, \Gamma^* \vdash e_2^*}$$

Second, specialization eliminates unnecessary branches of a `Receive` expression in case the specialized channel type found in the specialized channel type environment features less alternatives than the original type.

$$\frac{\begin{array}{c}\Theta(c) \leq [l_i(\tau_i) : \gamma_i]_{i=1}^{n} \qquad \Theta^*(c) \leq [l_i(\tau_i^*) : \gamma_i^*]_{i=1}^{m} \\ (\forall 1 \leq j \leq m)\ \Theta(c : \gamma_j), \Gamma(x_j : \tau_j) \vdash e_j \hookrightarrow \Theta^*(c : \gamma_j^*), \Gamma^*(x_j : \tau_j^*) \vdash e_j^* \end{array} \quad 1 \leq m \leq n}{\begin{array}{c}\Theta, \Gamma \vdash \texttt{Receive}\ c[l_i(x_i) \to e_i]_{i=1}^{n} \\ \hookrightarrow \Theta^*, \Gamma^* \vdash \texttt{Receive}\ c[l_i(x_i) \to e_i^*]_{i=1}^{m}\end{array}}$$

All the other translation rules for expressions only carry on the translation to subexpression in a compositional way leaving the actual expression context unchanged. For lack of space, we do not present the whole set of rules. The interested reader may find them elsewhere [18].

The second relation formally specifies a corresponding relation for definitions: a judgment $\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma' \hookrightarrow \Theta^*, \Gamma^* \vdash d \Rightarrow \Theta'^*, \Gamma'^*$ means that a typing

$\Theta^*, \Gamma^* \vdash d^* \Rightarrow \Theta'^*, \Gamma'^*$ is a valid specialization of an original definition with typing $\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma'$. There is one nontrivial specialization rule for definitions. A primitive operation where all arguments have singleton types specializes to a constant assignment.

$$\frac{\Gamma(x_i) = \Gamma^*(x_i) = i_i \qquad \mathtt{Op}(i_1, \ldots, i_n) \to i_0}{\begin{array}{c}\Theta, \Gamma \vdash x = \mathtt{Op}(x_1, \ldots, x_n) \Rightarrow \Theta, \Gamma(x : \mathbf{S}\{i_0 \,:\, b\}) \\ \hookrightarrow \Theta^*, \Gamma^* \vdash x = i_0 \Rightarrow \Theta^*, \Gamma^*(x : \mathbf{S}\{i_0 \,:\, b\})\end{array}}$$

Three rules affect the channel types found in the original program. Instead of introducing just one port type, γ, for each `NewPort` declaration, the typing of the specialized definition introduces two additional port types, γ' and γ'', one below the original type and one above it in the subtyping ordering. The idea is that the lower type γ' is the least upper bound of the server types connecting to the port whereas the upper type γ'' is the greatest lower bound of the (swapped) client types. The invariant for a type `Port` $(\gamma', \gamma, \gamma'')$ is $\gamma' \leq \gamma \leq \gamma''$.

$$\frac{\gamma' \leq \gamma \leq \gamma''}{\begin{array}{c}\Theta, \Gamma \vdash x = \mathtt{NewPort} \Rightarrow \Theta, \Gamma(x : \mathtt{Port}\ \gamma) \\ \hookrightarrow \Theta^*, \Gamma^* \vdash x = \mathtt{NewPort} \Rightarrow \Theta^*, \Gamma^*(x : \mathtt{Port}\ (\gamma', \gamma, \gamma''))\end{array}}$$

When specializing the beginning of a connection to a server (`Connect`), or when specializing a `Listen` to a channel, we use slice types to substitute the original channel type describing the original program behavior by specialized versions. In case of a `Listen`, a possible slice type γ_s substituted for γ must both follow the recursive structure of γ and also sit between the original channel type and the channel type describing all possible communication partners.

$$\frac{\Gamma(x) \geq \mathtt{Port}\ \gamma \qquad \Gamma^*(x) \geq \mathtt{Port}\ (\gamma', \gamma, \gamma'') \qquad \gamma'' \leq \gamma_s \trianglelefteq \gamma}{\Theta, \Gamma \vdash \mathtt{Listen}(x) \Rightarrow \Theta(c : \gamma), \Gamma \hookrightarrow \Theta^*, \Gamma^* \vdash \mathtt{Listen}(x) \Rightarrow \Theta^*(c : \gamma_s), \Gamma^*}$$

The situation for `Connect` commands is reversed. We again allow to use slice types as specialized versions of the original channel type γ. Here, the lower channel type used for splicing is the second channel type registered as port type, because we are now handling the other ends of the communication channels.

$$\frac{\Gamma(x) = \mathtt{Port}\ \gamma_0 \qquad \overline{\gamma_0} \leq \gamma \qquad \Gamma^*(x) \geq \mathtt{Port}\ (\gamma', \overline{\gamma_0}, \gamma'') \qquad \gamma' \leq \gamma_s \trianglelefteq \gamma}{\begin{array}{c}\Theta, \Gamma \vdash c = \mathtt{Connect}(x) \Rightarrow \Theta(c : \gamma), \Gamma \\ \hookrightarrow \Theta^*, \Gamma^* \vdash c = \mathtt{Connect}(x) \Rightarrow \Theta^*(c : \gamma_s), \Gamma^*\end{array}}$$

All the other translation rules again only carry on the translation to subexpression [18].

5.3 Properties of Translation

The first result states that valid translations of expressions and definitions imply that the original syntactic objects are already typeable.

Lemma 1.

(i) If $\Theta, \Gamma \vdash e \hookrightarrow \Theta^*, \Gamma^* \vdash e^*$*, then* $\Theta, \Gamma \vdash e$*.*
(ii) If $\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma' \hookrightarrow \Theta^*, \Gamma^* \vdash d^* \Rightarrow \Theta'^*, \Gamma'^*$*, then* $\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma'$*.*

The same property holds for results of valid translations, where $\widehat{\Gamma^*}$ denotes the type environment resulting from a Γ^* by removing lower subtypes and upper supertypes for each assigned port type.

Lemma 2.

(i) If $\Theta, \Gamma \vdash e \hookrightarrow \Theta^*, \Gamma^* \vdash e^*$*, then* $\Theta^*, \widehat{\Gamma^*} \vdash e^*$*.*
(ii) If $\Theta, \Gamma \vdash d \Rightarrow \Theta', \Gamma' \hookrightarrow \Theta^*, \Gamma^* \vdash d^* \Rightarrow \Theta'^*, \Gamma'^*$*,*
then $\Theta^*, \widehat{\Gamma^*} \vdash d^* \Rightarrow \Theta'^*, \widehat{\Gamma'^*}$*.*

We further show that valid translations induce two simulation properties. To be able to state those, an additional dynamic semantics for λ_{CS}, given as a reduction relation $\rightarrow$ on expressions, is assumed. The dynamic semantics for λ_{CS} has been given and invariance of typing under structural rearrangement of terms and type preservation have been proven elsewhere [18], but they are left here out due to space restrictions. Full type soundness cannot be proven because the type system is not strong enough to detect deadlocks.

Each evaluation step of a program can be simulated by zero or one evaluation step of translated program.

Lemma 3.
If $\emptyset, \emptyset \vdash e \hookrightarrow \emptyset, \emptyset \vdash e^*$ *and* $e \rightarrow e'$*, then* $\emptyset, \emptyset \vdash e' \hookrightarrow \emptyset, \emptyset \vdash e'^*$ *and* $e^* \rightarrow^{0,1} e'^*$*.*

For each evaluation step of a translated program, we find a finite number of evaluation steps for the original state simulating the single evaluation step.

Lemma 4.
If $\emptyset, \emptyset \vdash e \hookrightarrow \emptyset, \emptyset \vdash e^*$ *and* $e^* \rightarrow^{0,1} e'^*$*, then* $\emptyset, \emptyset \vdash e' \hookrightarrow \emptyset, \emptyset \vdash e'^*$ *and*
$e \rightarrow^+ e'$*.*

The translation is terminating because the specialized expression is smaller than the original expression. This fact can be determined by examination of the translation rules.

6 Related Work

Session types [9] have emerged as an expressive typing discipline for *heterogeneous*, bidirectional communication channels. In such a channel, each message may have a different type with the possible sequences of messages determined by the channel's session type. Such a type discipline subsumes typings for datagram communication as well as for homogeneous channels. Session types have been used to describe stream-based Internet protocols such as POP3 [9, 10].

Session types have also been proposed by Nierstrasz [19] for describing the behavior of active objects. The main idea is that a regular language on atomic

communication actions describes the sequence of messages on each channel. The session type specifies this language with a fixpoint expression. Each operation peels off the outermost action from the channel type so that each operation changes the channel's type.

Program specialization has been explored in a number of linguistic contexts from functional languages, logical and imperative languages, to object-oriented languages [15, 12, 24]. However, specialization for languages with concurrency has proved not very fruitful thus far and the main effort has been directed towards the removal of communication and nondeterminism. The most recent reference deals with a highly specialized area, concurrent constraint languages [7]. Marinescu and Goldberg [17] treat a simple CSP-like language and Gengler and Martel [11] consider the π-calculus. Unfortunately, both of them have technical problems. Hosoya et al [13] consider partial evaluation for HACL, an ML subset with channel-based communication.

Component adaptation by partial evaluation is not a new idea. Consel [2] proposed it for programs in general and Schultz [23] proposes a research agenda with desiderata for specialization of components. Interestingly, Schultz poses a question similar to the one we are investigating in the present paper in his conclusion: "Is it possible to unify the specialization needs that arise when the same component in a single application is used by several different components, possibly each through a unique interface?" We believe our techniques are applicable—with slight modifications—to this problem.

Bobeff and Noyé [1] also argue for a combination of partial evaluation and program slicing to perform component adaptation. Their approach requires the component provider to specify specialization scenarios and to perform the program analyses to support those scenarios. A component is then deployed as a component generator that creates specialized instances for each of the specified scenarios. While we are also relying on a program analysis, our goal is not a program generator but rather a specializer working on intermediate code.

The idea of specialization with respect to an inferred type is inspired by Hughes's type specializer [14]. However, our work is in a call-by-value setting with concurrency and communication whereas Hughes's work is in the context of the applied call-by-name lambda calculus.

None of the cited works guarantees terminating specialization and a guarantee that programs does not expand. Furthermore, in none of the works the specialization algorithm is type driven as is the case for our algorithm. The only type-driven specialization algorithms that we know of are Hughes's type specialization [14] and constructor specialization by Dussart and others [6]. But those specializers have different goals and target functional languages without concurrency.

Program slicing [16, 25] and application extraction [26] are transformations which are in spirit related to the effect of our specialization effort. The main difference is that we achieve the extraction with a pure specialization approach. While the relation between program specialization and program slicing has been

subject of some study [21], our work gives further indication of a deep relation between both techniques.

While the inspiration for our calculus comes from the work on calculi for communication as outlined in the introduction, the actual formulation is inspired by the capability calculus by Walker and others [27]. Our calculus may be viewed as a specialized instance of the capability calculus, first, with respect to the typing discipline (simple types instead of polymorphism) and, second, with respect to the application area, channel-based communication.

7 Conclusion

We have defined a framework for component adaption and specialization based on an intermediate language with concurrent functional processes. The framework allows removal of dead code by specializing conditionals. We have proved the soundness of the specialization algorithm. The specialization algorithm terminates always and never expands a source program beyond its original size.

References

1. Gustavo Bobeff and Jacques Noyé. Molding components using program specialization techniques. In *WCOP 2003, Eighth International Workshop on Component-Oriented Programming*, Darmstadt, Germany, July 2003.
2. Charles Consel. Program adaptation based on program transformation. *ACM Computing Surveys*, 28(4es):164, 1996.
3. Charles Consel, editor. *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Amsterdam, The Netherlands, June 1997. ACM Press.
4. Clemens Czyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
5. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, number 1110 in Lecture Notes in Computer Science, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.
6. Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant constructor specialization. In William Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 54–63, La Jolla, CA, USA, June 1995. ACM Press.
7. Sandro Etalle and Maurizio Gabbrieli. Partial evaluation of concurrent constraint languages. *ACM Comput. Surv.*, 30(3es):11, 1998.
8. Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtying revealed. *Journal of Functional Programming*, 12(6):511–548, November 2002.
9. Simon Gay and Malcolm Hole. Types and subtypes for client-server interactions. In Doaitse Swierstra, editor, *Proceedings of the 1999 European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 74–90, Amsterdam, The Netherlands, April 1999. Springer-Verlag.
10. Simon Gay, Vasco Vasconcelos, and Antonio Ravara. Session types for inter-process communication. Technical Report TR-2003-133, Department of Computing Science, University of Glasgow, 2003.

11. Marc Gengler and Matthieu Martel. Self-applicable partial evaluation for the pi-calculus. In Consel [3], pages 36–46.
12. John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors. *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in Lecture Notes in Computer Science, Copenhagen, Denmark, 1999. Springer-Verlag.
13. Haruo Hosoya, Naoki Kobayashi, and Akinori Yonezawa. Partial evaluation scheme for concurrent languages and its correctness. In L. Bougé et al., editors, *Euro-Par'96 - Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 625–632, Lyon, France, 1996. Springer-Verlag.
14. John Hughes. Type specialisation for the λ-calculus; or, a new paradigm for partial evaluation based on type inference. In Danvy et al. [5], pages 183–215.
15. Neil Jones, Carsten Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
16. Naoki Kobayashi. Useless code elimination and programm slicing for the pi-calculus. In Atsushi Ohori, editor, *Proceedings of the First Asian Symposium on Programming Languages and Systems*, number 2895 in Lecture Notes in Computer Science, pages 55–72, Beijing, China, November 2003. Springer-Verlag.
17. Mihnea Marinescu and Benjamin Goldberg. Partial-evaluation techniques for concurrent programs. In Consel [3], pages 47–62.
18. Matthias Neubauer and Peter Thiemann. Protocol specialization. Technical Report 212, Institut für Informatik, University of Freiburg, Germany, August 2004.
19. Oscar Nierstrasz. Regular types for active objects. In *Proceedings OOPSLA '93*, pages 1–15, October 1993.
20. Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proc. of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 376–385. IEEE Computer Society Press, 1993.
21. Thomas Reps and Todd Turnidge. Program specialization via program slicing. In Danvy et al. [5], pages 409–429.
22. Ulrik Schultz, Julia Lawall, Charles Consel, and Gilles Muller. Toward automatic specialization of Java programs. In *13th European Conference on Object-Oriented Programming (ECOOP '99)*, Lisbon, June 1999. Springer-Verlag.
23. Ulrik P. Schultz. Black-box program specialization. In *WCOP'99, Fourth International Workshop on Component-Oriented Programming*, Lisbon, Portugal, June 1999.
24. Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. Automatic program specialization for java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):452–499, 2003.
25. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
26. Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical extraction techniques for Java. *ACM Transactions on Programming Languages and Systems*, 24(6):625–666, November 2002.
27. David Walker, Carl Crary, and Greg Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.