From Sequential Programs to Multi-Tier Applications by Program Transformation

Matthias Neubauer Peter Thiemann
Institut für Informatik, Universität Freiburg, Germany
{neubauer,thiemann}@informatik.uni-freiburg.de

ABSTRACT

Modern applications are designed in multiple tiers to separate concerns. Since each tier may run at a separate location, middleware is required to mediate access between tiers. However, introducing this middleware is tiresome and error-prone.

We propose a *multi-tier calculus* and a *splitting transformation* to address this problem. The multi-tier calculus serves as a sequential core programming language for constructing a multi-tier application. The application can be developed in the sequential setting. Splitting extracts one process per tier from the sequential program such that their concurrent execution behaves like the original program.

The splitting transformation starts from an assignment of primitive operations to tiers. A program analysis determines communication requirements and inserts remote procedure calls. The next transformation step performs resource pooling: it optimizes the communication behavior by transforming sequences of remote procedure calls to a stream-based protocol. The final transformation step splits the resulting program into separate communicating processes.

The multi-tier calculus is also applicable to the construction of interactive Web applications. It facilitates their development by providing a uniform programming framework for client-side and server-side programming.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Distributed Programming*;
D.1.2 [**PROGRAMMING TECHNIQUES**]: Automatic Programming—*Program Transformation*; F.3.2 [**LOGICS AND MEANINGS OF PROGRAMS**]: Semantics of Programming Languages—*Operational semantics*

General Terms

Design, Languages, Theory

POPL'05, January 12–14, 2005, Long Beach, California, USA.

Keywords

Type Systems, Concurrency, Application Partioning

1. INTRODUCTION

Building a modern application is no longer a task of programming an isolated machine. In many cases, the application has a distributed multi-tier architecture. It accesses data on one or more database servers, its business logic runs on an application server, the presentation logic is deployed on a web server, and the user interface runs on a web browser. Software for each tier is developed separately with application-specific communication interfaces between the tiers. The designers create these interfaces and the programmers implement them using middleware. Hence, on top of their usual coding and testing activity, programmers are also burdened with the quirks of distribution: testing becomes more difficult and flaws in middleware code—often created by third-party programmers—are potentially an additional source of errors in the final application.

The programmer's task can be alleviated by a programming framework that automatically prepares the program for the physical distribution. Thus, the main development activity takes place in the simpler, non-distributed setting. Only integration testing and performance testing (and, of course, final deployment of the application) involves the distributed production setting.

The present paper proposes such a programming framework consisting of a calculus, a static analysis that performs an assignment of code to locations, and a range of program transformations that automatically insert communication primitives, perform resource pooling, and finally split the program into separate processes to run at separate locations. Locations in our sense are used as an abstraction over the different physical or logical places where tiers of the final application are supposed to reside.

As a starting point, we assume that the application consists of sequential programs that run concurrently on a single server. Each program runs independently on behalf of a particular client. The programs do not communicate explicitly among themselves but they access and modify common resources on the server. The task of the transformation is thus to split a sequential program into pieces that run independently from each other on different hosts and to equip them with communication primitives so that the sequential semantics of the original program is preserved.

Each operation used in the sequential program carries an annotation that indicates whether the operation is location independent. A location independent operation must not

have side effects and it must not depend on data stored at a particular location. For example, any operation on a primitive datatypes is location independent whereas accessing or modifying a reference is not. Location dependent operations carry a location annotation that indicates where the operation can safely take place. The interplay of all location annotations drives the static analysis of the program. Operations on different locations must not interfere with each other. For example, database queries must run on the application server because the database connection is a reference that does not make sense outside of the application server. In contrast, GUI operations run on the client because the GUI objects are not present on the server.

In principle, the result of the the splitting transformation is directly applicable to annotated programs but it would lead to programs sprinkled with short communications. Since repeated connection establishment is expensive, a separate program transformation performs resource pooling by reusing established connections between locations.

We describe the overall algorithm abstractly in terms of transformation steps in a program calculus. The calculus is based on the lambda calculus equipped with synchronous communication primitives. It is inspired by Gay and Hole's calculus of session types [2]. Their calculus is a simply typed variant of the π-calculus with subtyping and models asynchronous communication via ports and channels. Channels transmit heterogeneous data as prescribed by a session type.

Unlike Gay and Hole, we have chosen a lambda calculus because it is trivial to embed sequential programs in it. In particular, our variant is based on a simply-typed, applied call-by-value lambda calculus in A-normal form[1]. Communication in our calculus is stream-based (ports and channels) with similar operations as in the calculus of session types. A-normal form simplifies the definition of the static and dynamic semantics and has been used as an intermediate language in several compilers.

The result of the transformation is a program with explicit uses of typed communication primitives analogous to the socket-based communication paradigm [15]. An implementation would map the communication primitives to the communication architecture provided by the chosen middleware. Alternatively, a direct implementation in terms of the socket API is possible.

A web application is a special case of a multi-tier application where one or more tiers run on a web browser. Our framework is applicable to this case given a suitable mediator that implements channels on top of HTTP. In this special case, the splitting transformation yields a client-side slice and a server-side slice of the application. The client's slice may be implemented by applets and the server's slice with an arbitrary server-side scripting technology.

The main contribution of the paper is the partitioning algorithm. Its presentation consists of three steps:

- We specify a location analysis that infers required communications between abstract locations. The analysis is stated in terms of an annotated simple type system with subtyping. It is proved correct with respect to a location-aware operational semantics using a type-soundness argument.

- We present a suite of typed transformation rules that merge several communications between identical partners into larger sections with stream-based communication. The rules are proven correct with respect to the location-aware semantics extended with synchronous communication.

- A splitting transformation finally extracts concurrently running processes from the transformed expression. This transformation is also proved correct. The target calculus is a language with channel-based communication. Its type system is a novel variant of the system of session types [2].

```
let rec
 show_topic (db, tid, tpc) =
   let stmt  = string_append_4
                 ("SELECT * FROM messages",
                  " WHERE tid='", tid, "'") in
   let msgs  = sql_exec (db, stmt) in
   let hdr   = string_append ("Topic ", tpc) in
   let frame = gui_create_frame (hdr) in
   let _     = show_rows (db, frame, msgs) in
   gui_show_frame (frame)
 and
 show_rows (db, frame, msgs) =
   let b = sql_cursor_has_more_elements
             (db, msgs) in
   if b then
      let row = sql_cursor_get_next_row
                  (db, msgs) in
      let cts = sql_cursor_get_column
                  (db, row, "contents") in
      let lab = gui_create_label (cts) in
      let _   = gui_frame_add (frame, lab) in
      show_rows (frame, msgs)
   else
      ()
in ...
```

Figure 1: Original sequential program

In the rest of the paper, we first motivate our suite of transformations with an excerpt of a client-server application in Section 2. Section 3 introduces the source calculus, a simply-typed, linearized, call-by-value lambda calculus. Section 4 develops the annotated type system that specifies location analysis for the source calculus. It formalizes its static and dynamic semantics and proves type soundness. Section 5 introduces the multi-tier calculus which extends the source calculus with explicit channel-based communication primitives. Section 6 defines the transformation steps to implement connection pooling and Section 7 presents the splitting transformation. Section 8 discusses related work, and Section 9 concludes.

2. A TINY MULTI-TIER APPLICATION

This section introduces an example application, a message board, and shows and discusses the results of the location analysis and the splitting transformation. For concreteness, the example is programmed in Ocaml [9] and is easy to translate to the formal calculus introduced in Section 3.

A message board application enables its users to view and post messages attached to certain topics on a centrally accessible virtual blackboard. A message either starts a new topic or it responds to a message in an existing topic.

The application draws on at least two kinds of resources that usually reside in different physical locations: a central database server and a windowing environment handling user

```
let rec
  show_topic (db[S], tid[S], tpc[S]) =
    let stmt  = string_append_4[S]
                  ("SELECT * FROM messages",
                   " WHERE tid='", tid, "'") in
    let msgs  = sql_exec[S] (db, stmt) in
    let tpc   = trans[S -> C] tpc in
    let hdr   = string_append[C] ("Topic ", tpc) in
    let frame = gui_create_frame[C] (hdr) in
    let _     = show_rows[S,C] (db, frame, msgs) in
    gui_display_frame[C] (frame)
  and
  show_rows (db[S], frame[C], msgs[S]) =
    let b = sql_cursor_has_more_elements[S]
              (db, msgs) in
    let b = trans[S -> C] b in
    if b then
       let row = sql_cursor_get_next_row[S]
                   (db, msgs) in
       let cts = sql_cursor_get_column[S]
                   (db, row, "contents") in
       let cts = trans[S -> C] cts
       let lab = gui_create_label[C] (cts) in
       let _   = gui_frame_add[C] (frame, lab) in
       show_rows[S,C] (frame, msgs)
    else
       ()[S,C]
in ...
```

Figure 2: Program after location analysis (fat client)

interactions. The database is accessible via a textual SQL interface and the windowing environment provides the usual GUI widgets. The database contains one relation `messages` where each tuple describes one message (topic, contents, author, etc). Administrative fields include a topic id (`tid`) and a flag indicating whether the message starts a new topic.

Figure 1 shows two functions taken from the program.[1] `Show_topic` implements a basic operation of the message board, the display of a list of messages for a given topic. To do so, it first queries the database using an existing database connection, `db`, for a given topic id, `tid`, and presents the content fields of the resulting message set in a new GUI frame. It relies on `show_rows` to populate the GUI frame with a text label for each message with that topic id.

The code relies on several library functions to access the database and to perform GUI operations. In particular, the function `sql_exec` returns a cursor to the result of an SQL query, the functions `sql_cursor_...` access the relation underlying the cursor, and the `gui_...` functions create and manipulate appropriate GUI widgets.

Not all library functions are location independent. The `sql_...` operations are not because all take the database connection `db` as a parameter. The annotation `[S]` indicates that the connection is only available on the server, hence the location analysis prescribes that all `sql_...` operations must take place on the server, too. In contrast, the GUI widgets can be created anywhere (`gui_create_frame`, `gui_create_label`, and `gui_frame_add`), but there is also one operation, `gui_display_frame`, that must run on the client machine (indicated by the `[C]` annotation). Operations like `string_append_...` are location independent and may execute anywhere.

The location analysis assigns to each operation a set of locations where the operation must take place. Starting from the operations with a fixed initial assignment, location information is propagated through the program, and an "implicit" communication `y = trans`$^{[A \to B]}$ `x` is inserted whenever data `x` available at A is required—but not yet available—at location B. The outcome of the analysis is determined by a propagation strategy. Depending on this strategy, the analysis may produce fat clients or thin clients and it may choose to duplicate operations at multiple locations to avoid the overhead of transmitting their results. Figure 2 shows one possible outcome of the analysis where the GUI widgets are constructed on the client.

At this point, the program in Figure 2 may already be split into a client process and a server process. However, doing so would be inefficient because each `trans` builds a connection between client and server to transmit one data item. Our joining transformation addresses this inefficiency. It joins adjacent communications, thus switching from a message-based style to a stream-based style. The main idea of the joining transformation is to introduce explicit channels with `open` and `close` operations and then float `open` operations towards `close` operations in hopes of merging them. The transformation has to go into some complication to deal smoothly with conditionals and with recursion.

Finally, the splitting transformation extracts from the annotated program one slice for each location. Running all slices in parallel is equivalent to running the original program. Figure 3 contains the final program for the example. It contains two processes which share the communication port `p`[2]. Each process has the same structure because it is essentially a slice of the original program. The server process first opens the server end of a channel through port `p` using `listen`. Its slice of `show_rows` first sends a boolean indicating whether further data is following. If that is the case, it sends the first tuple and attempts to process the next tuple recursively. Otherwise, `show_rows` exits and the channel is closed. The client process performs exactly the converse communications: it first opens the client end of the channel and then receives data as it is sent. Technically, the communication is typed using a *session type* of the form

$$(\overline{string}, \mu\beta.(\overline{boolean}, \langle \texttt{true} \to (\overline{tuple}, \beta) \mid \texttt{false} \to \varepsilon\rangle)).$$

The type describes the possible sequences of communication events on a channel: first read a *string*; then repeatedly read a *boolean*; then either read a *tuple* and continue or close the channel. The latter choice depends on the communication labels `true` and `false` which are used in the type applications `chan{true}` and `chan{false}`. They indicate on the type level, which branch of the conditional is taken. Hence, `true` and `false` are *not* values, but rather communication labels which have a status like record labels.

In the explanation above, we have glossed over intermediate technical steps that introduce additional infrastructure in the term. For example, one step introduces channels with `open` and `close` operations. We defer closer discussion of the intermediate steps to subsequent sections (Section 6 and 7).

[1] In Ocaml, the left-hand side of a `let` is a pattern, which has to match the value of the right-hand side. The pattern `_` is a wildcard pattern that matches any value.

[2] The `newPort` expression will be written νp in the calculus.

```
let p = newPort in

// server process                                    ||  // client process
let rec                                                  let rec
  show_topic (db, tid, tpc) =                              show_topic () =
    let chan   = listen (p) in                               let chan  = connect (p) in
    let stmt   = string_append_4                             let tpc   = recv (chan) in
                   ("SELECT * FROM messages",                let hdr   = string_append ("Topic ", tpc) in
                    " WHERE tid='", tid, "'") in             let frame = gui_create_frame (hdr) in
    let msgs   = sql_exec (db, stmt) in                      let _     = show_rows [chan] (frame) in
    let          send chan (tpc) in                          let         close (chan) in
    let _      = show_rows [chan] (db, msgs) in              gui_display_frame (frame)
    let          close (chan) in                           and
    ()                                                     show_rows [chan] (frame) =
  and                                                        let b = recv (chan) in
  show_rows [chan] (db, msgs) =                              if b then
    let b = sql_cursor_has_more_elements                        let       chan{true} in
              (db, msgs) in                                     let cts = recv (chan) in
    let     send chan (b) in                                    let lab = gui_create_label (cts) in
    if b then                                                   let _   = gui_frame_add (frame, lab) in
       let       chan{true} in                                  show_rows [chan] (frame)
       let row = sql_cursor_get_next_row                     else
                   (db, msgs) in                               let            chan{false} in
       let cts = sql_cursor_get_column                         ()
                   (db, row, "contents") in                in ...
       let       send chan (cts) in
       show_rows [chan] (db, msgs)
    else
       let            chan{false} in
       ()
in ...
```

Figure 3: Program split up into separate processes

3. SOURCE CALCULUS

The underlying programming model is the simply-typed call-by-value lambda calculus with integers, primitive operations, and recursion. To simplify matters, the calculus λ_A is an intermediate language. Its syntax is given by

$x \in \text{Var}, i \in \mathbf{Z}$
Expressions
$e \quad ::= \quad \texttt{halt} \mid \texttt{let}\ d\ \texttt{in}\ e \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid x\,(\tilde{x})$
Statements
$d \quad ::= \quad x = \texttt{pfun}(\tilde{x}) \mid x = \texttt{op}(\tilde{x}) \mid \texttt{rec}\ \{\, r \,\}$
$r \quad ::= \quad \varepsilon \mid x(\tilde{x}) = e \mid r, r$

A λ_A expression is a sequence of let-bound statements, d, ending either in a `halt` instruction, a conditional, or a jump with parameters. A statement either performs a primitive function, an operation, or introduces a set of mutually recursive jump labels.

All argument subterms in expressions and statements are restricted to variables. A primitive function $\texttt{pfun}(\tilde{x})$ is free of side effects. Nullary primitive functions serve as constants. An operation $\texttt{op}(\tilde{x})$ has unspecified side effects. Arguments and results of primitive functions and operations are restricted to first-order values. The notation $\tilde{x}$ stands for the sequence $x_1, \ldots, x_n$ where n derives from the context.

We refrain from stating the type system or a semantics for the source calculus because both are straightforward to derive from the definitions (for extended calculi) in the subsequent sections.

4. INTRODUCING LOCATIONS

A λ_A expression describes a computation at one location. This section extends λ_A to λ'_A, which can express that certain computations run on specific locations (*i.e.*, tiers). In λ'_A there is still only one program executing with one centralized locus of control. However, at each location only a subset of the program's effect will be visible. Values of variables are only available on a subset of locations, some operations are only available at specific locations, and data must be moved explicitly from one location to another. Syntactically, there is one new statement and one modification:

$$d \quad ::= \quad \cdots \mid x = \texttt{trans}^{[A \rightsquigarrow B]}\ x \mid x = \texttt{op}^A(\tilde{x})$$

The statement $x = \texttt{trans}^{[A \rightsquigarrow B]}\ y$ transmits the value of y from location A to location B, where $A, B \in \mathcal{N}$, a finite set of locations. The value of y must be available at A before executing the statement. Afterwards, the value is available through x at location B as well as at all locations at which it was available through y. Typically, the `trans` statement is used with $x = y$ so that it just extends the availability of x to include B.

Only base values can be transmitted between locations. Functions and pointers (references) cannot be transmitted.

The statement $\texttt{op}^A(\tilde{x})$ performs an operation that has a side effect on location A and thus is only available on that location. The values of $\tilde{x}$ must be available at A before executing the statement and the result of the operation will be available only at A afterwards. The side effect of $\texttt{op}^A(\tilde{x})$ is visible on A but not on any other location.

In the rest of this section, we first introduce the dynamic semantics of λ'_A in terms of a labeled transition system. Next, we present a type system that tracks the location of values in addition to their actual type. The type system extends the system of simple types with location annotations and annotation subtyping. Finally, we prove type soundness for this system.

4.1 Dynamic Semantics

The dynamic semantics of λ'_A is defined by a small-step transition system. The intermediate states of the system require an extension of the syntax, as usual. There are two kinds of computed values in the system:

$$v \quad ::= \quad \texttt{val}(i; N) \mid \texttt{fun}(\texttt{rec}\ \{\, r \,\}; f)$$

A value can be either a constant i with a set of locations N, which indicates where the value is available, or it can be a function consisting of a function name f and a set r of mutually recursive function definitions (where f selects one of the function of r). Functions do not carry a set of locations because they are assumed to be available at all locations.

The original expressions and statements are extended to admit values wherever bound occurrences of variables are allowed in the original syntax.

Evaluation steps produce traces of observations. Traces of observations, $w \in l^*$, are words over observations, l:

$$l \quad ::= \quad \texttt{halt} \mid i_0 = \texttt{op}^A(\tilde{i})$$

where `halt` is intended to register halting expressions, and $i_0 = \texttt{op}^A(\tilde{i})$ to register a particular call pattern of operation $\texttt{op}()$ executed on location A. With $locs(w)$, we denote the set of all locations, A, found all the observations of operations, $i_0 = \texttt{op}^A(\tilde{i})$, occurring in the a trace w. Throughout the paper, we specify reductions as families of relations indexed by a trace, $\xrightarrow{w}$.

Figure 4 defines several notions of reduction for λ'_A. The relation $e \xrightarrow{w}_{\text{core}} e'$ is the core reduction relation which remains constant for the rest of this work. It describes evaluation steps that transform e to e' producing no side effect. The $\xrightarrow{w}_{t1}$ reduction deals with the `Trans` statement. The relation $e \xrightarrow{w}_{\text{op}} e'$ describes operations with side effects. With $\xrightarrow{w}_{\text{op}^A}$, we denote reductions of operations on location A. In addition, we make use of the following combined relations $\xrightarrow{w}_{\text{imp}} = \xrightarrow{w}_{\text{core}} \cup \xrightarrow{w}_{\text{op}}$, $\xrightarrow{w}_{\lambda'_{A,pure}} = \xrightarrow{w}_{\text{core}} \cup \xrightarrow{w}_{t1}$, $\xrightarrow{w}_{\lambda'_A} = \xrightarrow{w}_{\text{imp}} \cup \xrightarrow{w}_{t1}$.

The rule for primitive functions substitutes the result of the function as determined by $\delta_{\texttt{pfun}}$ (a partial function) in the rest of the term. The result is available at those locations where all arguments are available. A primitive operation only happens at its designated location and leaves its trail in the trace. Operations behave nondeterministically so they are modeled by a *relation* δ. This choice may seem odd at first. However, the model must include the possibility that on each location further processes run in parallel to the program. That is, we cannot model side effects by passing a state for each location because this state may change between two operations of our program due to effects from another process. The nondeterministic model avoids passing state explicitly and encompasses arbitrary effects from other processes.

The definition of a set of mutually recursive labels substitutes a function value for each defined label. The function value consists of the label identifier paired with the definition group. The conditional dispatches evaluation to the true or the false branch, as usual. A function call extracts the selected function definition from the definition group and substitutes the arguments into that function's body. Since the function's body may refer to other functions in the definition group, it is wrapped in a new `let` with the definitions of the same definition group.

$\boxed{\vdash' \tau \leq \tau'}$

$$\frac{N_1 \supseteq N_2}{\vdash' (b, N_1) \leq (b, N_2)} \qquad \frac{N_1 \supseteq N_2 \quad \vdash' \tilde{\tau}_2 \leq \tilde{\tau}_1}{\vdash' \tilde{\tau}_1 \xrightarrow{N_1} 0 \leq \tilde{\tau}_2 \xrightarrow{N_2} 0}$$

$\boxed{\Gamma \vdash'_a x : \tau}$

$$\frac{\Gamma \vdash'_a x : \tau \quad \tau \leq \tau'}{\Gamma \vdash'_a x : \tau'} \qquad \Gamma \vdash'_a x : \Gamma(x)$$

Figure 5: Subtyping and argument rules for λ'_A

4.2 Static Semantics

The type system for λ'_A has to model two facets of each value: its shape and the locations where it is available. It keeps track of the shape using an underlying simply-typed system and adds location annotations and effects to keep track of the locations. Annotation subtyping produces precise analysis results [16].

An annotated type, τ, is either a base type b paired with a set of locations N or a function type with a location set N as latent effect. The meaning of an annotation N is that the associated data item *must* be present at all locations in N. The effect on a function type indicates the set of locations that may execute operations when the function is applied.

$$\tau ::= (b, N) \mid \tilde{\tau} \xrightarrow{N} 0 \qquad\qquad N \subseteq \mathcal{N}$$

There are three typing judgments.

- $\Gamma \vdash' e \,!\, N$ states that e uses variables in Γ correctly and may perform operations at locations N.
- $\Gamma \vdash' d \Rightarrow \Gamma' \,!\, N$ states that d transforms Γ to Γ' and may perform operations at N.
- $\Gamma \vdash'_a x : \tau$ infers the type for an argument position.

Subtyping in the system is structural and induced solely by the location annotations. Subtyping—as formalized by the judgment $\vdash' \tau \leq \tau'$—expresses that a value that is present at location set N can substitute for a value that is only expected at a subset $N' \subseteq N$. Also, the effect annotation of a subtype must be included in the annotation of the supertype. Figures 5 and 6 contain the subtyping rules and the typing rules for expressions and statements.

The `halt` expression has neither requirements nor does it perform an effect at any location. The effect of a let statement is the union of the effects of the statement and the body expression. A conditional needs only be executed on a location if one of the branches contains code to be executed on that location. Hence, the value of the condition needs only be available on the locations mentioned in the effect of the branches. A function application unleashes the latent effect of the function.

The rules for primitive functions and operations reflect their operational semantics but use subtyping to intersect the location sets implicitly. A recursive label is implicitly available at all locations. The `trans` statement copies the value of y (which must be available at location A) to location B and binds it to x with appropriately changed type.

$$\begin{array}{rcll}
\texttt{let } x = \texttt{pfun}(\dots \texttt{val}(i_j; N_j) \dots) \texttt{ in } e & \xrightarrow{\varepsilon}_{\text{core}} & e[x \mapsto \texttt{val}(i; \bigcap N_j)] & \textbf{if } i = \delta_{\texttt{pfun}}(\tilde{i}) \wedge \bigcap N_j \neq \emptyset \\
\texttt{let rec } \{\, f_i(\tilde{x}_i) = e_i \,\}_{i=1}^n \texttt{ in } e & \xrightarrow{\varepsilon}_{\text{core}} & e[f_j \mapsto \texttt{fun}(\texttt{rec } \{\, f_i \dots \}_{i=1}^n; f_j)]_{j=1}^n & \\
\texttt{if } (\texttt{val}(0; N)) \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\varepsilon}_{\text{core}} & e_2 & \textbf{if } N \neq \emptyset \\
\texttt{if } (\texttt{val}(i; N)) \texttt{ then } e_1 \texttt{ else } e_2 & \xrightarrow{\varepsilon}_{\text{core}} & e_1 & \textbf{if } i \neq 0 \textbf{ and } N \neq \emptyset \\
(\texttt{fun}(\texttt{rec } \{\, f_i(\tilde{x}_i) = e_i \,\}_{i=1}^n; f_j))\,(\tilde{v}) & \xrightarrow{\varepsilon}_{\text{core}} & \texttt{let rec } \{\, f_i(\tilde{x}_i) = e_i \,\}_{i=1}^n \texttt{ in } e_j[\tilde{x}_j \mapsto \tilde{v}] & \\
\texttt{let } x = \texttt{op}^A(\dots \texttt{val}(i_j; N_j) \dots) \texttt{ in } e & \xrightarrow{i=\texttt{op}^A(\tilde{i})}_{\text{op}} & e[x \mapsto \texttt{val}(i; \{A\})] & \textbf{if } i = \texttt{op}(\tilde{i}) \in \delta \wedge A \in \bigcap N_j \\
\texttt{let } x = \texttt{trans}^{[A \rightsquigarrow B]}\,(\texttt{val}(i; N)) \texttt{ in } e & \xrightarrow{\varepsilon}_{t1} & e[x \mapsto \texttt{val}(i; N \cup \{B\})] & \textbf{if } A \in N
\end{array}$$

Figure 4: Reduction rules for λ'_A

$\boxed{\Gamma \vdash' e \,!\, N}$

$$\Gamma \vdash' \texttt{halt} \,!\, \emptyset$$

$$\frac{\Gamma \vdash' d \Rightarrow \Gamma' \,!\, N_d \qquad \Gamma' \vdash' e \,!\, N_e}{\Gamma \vdash' \texttt{let } d \texttt{ in } e \,!\, N_d \cup N_e}$$

$$\frac{\Gamma \vdash'_a x : (b, N) \qquad N_1, N_2 \subseteq N \qquad \Gamma \vdash' e_1 \,!\, N_1 \qquad \Gamma \vdash' e_2 \,!\, N_2}{\Gamma \vdash' \texttt{if } x \texttt{ then } e_1 \texttt{ else } e_2 \,!\, N_1 \cup N_2}$$

$$\frac{\Gamma \vdash'_a x_1 : \tilde{\tau} \xrightarrow{N} 0 \qquad \Gamma \vdash'_a \tilde{x}_2 : \tilde{\tau}}{\Gamma \vdash' x_1\,(\tilde{x}_2) \,!\, N}$$

$\boxed{\Gamma \vdash' d \Rightarrow \Gamma' \,!\, N}$

$$\frac{(\forall 1 \leq i \leq n)\ \Gamma \vdash'_a x_i : (b, N)}{\Gamma \vdash' x = \texttt{pfun}(x_1, \dots, x_n) \Rightarrow \Gamma(x : (b, N)) \,!\, \emptyset}$$

$$\frac{(\forall 1 \leq i \leq n)\ \Gamma \vdash'_a x_i : (b, \{A\})}{\Gamma \vdash' x = \texttt{op}^A(x_1, \dots, x_n) \Rightarrow \Gamma(x : (b, \{A\})) \,!\, \{A\}}$$

$$\frac{(\forall 1 \leq j \leq n)\ \Gamma(f_i : \tilde{\tau}_i \xrightarrow{N_i} 0)_{i=1}^n (\tilde{x}_j : \tilde{\tau}_j) \vdash' e_j \,!\, N_j}{\Gamma \vdash' \texttt{rec } \{\, f_i(\tilde{x}_i) = e_i \,\}_{i=1}^n \Rightarrow \Gamma(f_i : \tau_i \xrightarrow{N_i} 0)_{i=1}^n \,!\, \emptyset}$$

$$\frac{\Gamma \vdash'_a y : (b, N) \qquad A \in N}{\Gamma \vdash' x = \texttt{trans}^{[A \rightsquigarrow B]}\ y \Rightarrow \Gamma(x : (b, N \cup \{B\})) \,!\, \emptyset}$$

Figure 6: Typing rules for λ'_A

4.3 Properties of λ'_A

To connect λ_A with λ'_A requires an erasure function $|\cdot|$ that maps an λ'_A expression to an λ_A expression by forgetting about locations and removing all $\texttt{trans}^{[A \rightsquigarrow B]}\ x$ statements. On types, the erasure function strips away all location sets and all effects. Erasure has the following properties with respect to the underlying unannotated calculus whose typing judgment is indicated with $\vdash_0$.

LEMMA 1. *1. If $\Gamma' \vdash' e' \,!\, N'$ then $|\Gamma'| \vdash_0 |e'|$.*

2. If $\Gamma \vdash_0 e$ then there exists some Γ', e', and N' such that $\Gamma' \vdash' e' \,!\, N'$ and $|\Gamma'| = \Gamma$ and $e' = |e|$.

The expression e' constructed from e in the second part is a *completion of* e. The existence of completions is shown by inserting transmission statements to every location after every statement. Completions are not uniquely defined, in general. For example, consider $\mathcal{N} = \{S, C\}$ and operations s and c which are only available on S and C, respectively.

source	first completion	second completion
$x = s_1^S()$	$x = s_1^S()$	$x = s_1^S()$
$y = x + 1$	$x = \texttt{trans}^{[S \rightsquigarrow C]}\ x$	$y = x + 1$
$c^C(y)$	$y = x + 1$	$y = \texttt{trans}^{[S \rightsquigarrow C]}\ y$
halt	$c^C(y)$	$c^C(y)$
	halt	halt

There are many further completions that perform additional useless transmissions. The example also shows that there is no obvious notion of optimality or minimality for completions. The first completion may perform fewer operations on S whereas the second performs fewer operations on C. It is not clear which is preferable without more information about C and S.

For the moment, we leave this question to an implementation of the analysis in future work. The implementation will make its choice based on formalized preferences between locations. The implementation may even introduce redundancy by performing a computation at more than one location. This choice trades communication with computation.

It is routine to prove type soundness for the calculus λ'_A. That is, evaluation of a typed closed expression either reaches `halt` after a finite number of steps or it keeps reducing forever. The typing rules for values and the intermediate states are straightforward.

LEMMA 2 (TYPE PRESERVATION). *Suppose $\emptyset \vdash' e \,!\, N$ and $e \xrightarrow{w}_{\lambda'_A} e'$ then $\emptyset \vdash' e' \,!\, N'$, $N' \subseteq N$, and $locs(w) \subseteq N$.*

LEMMA 3 (PROGRESS). *If $\emptyset \vdash' e!N$ then $e = \texttt{halt}$ or there exists e' such that $e \xrightarrow{w}_{\lambda'_A} e'$ and $locs(w) \subseteq N$.*

Furthermore, we define the reflexive transitive closure $\xrightarrow{w}{}^*$ of the evaluation relation as follows:

$$e \xrightarrow{\varepsilon}{}^* e \qquad \frac{e \xrightarrow{w'} e' \quad e' \xrightarrow{w''}{}^* e''}{e \xrightarrow{w'w''} e''}$$

Type soundness follows by straightforward induction.

THEOREM 1. *If $\emptyset \vdash' e \,!\, N$ then either $e \xrightarrow{w}{}^*_{\lambda'_A} \texttt{halt}$ and $locs(w) \subseteq N$ or, for each e', if $e \xrightarrow{w'}{}^*_{\lambda'_A} e'$ then there exists e'' such that $e' \xrightarrow{w'}_{\lambda'_A} e''$ and $locs(w'') \subseteq N$.*

$$
\begin{array}{lcl}
l & \in & \text{Label}, x \in \text{Var}, c \in \text{ChannelVar} \\
\text{Statements} & & \\
d & ::= & x = \texttt{pfun}(\tilde{x}) \mid x = \texttt{op}^{A}(\tilde{x}) \mid \texttt{rec}\ \{\, r \,\} \\
& \mid & c = \texttt{open}^{[A \rightsquigarrow B]} \mid c = \texttt{openP}^{[A \rightsquigarrow B]}\ (p) \\
& \mid & \texttt{close}^{[A \rightsquigarrow B]}\ (c) \mid x = \texttt{trans}^{[A \rightsquigarrow B]}\ c(x) \mid c\{l\} \\
r & ::= & \varepsilon \mid x[\tilde{c}](\tilde{x}) = e\ ;\ r \\
\text{Expressions} & & \\
e & ::= & \texttt{halt} \mid \texttt{let}\ d\ \texttt{in}\ e \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid & x\,[\tilde{c}]\,(\tilde{x}) \mid \nu p.e \\
\text{Types and Type Environments} & & \\
\tau & ::= & (b, N) \mid [\tilde{\gamma}]\tilde{\tau} \xrightarrow{N} 0 \\
\Gamma & ::= & \emptyset \mid \Gamma(x : \tau) \\
\text{Session Types and Session Type Environments} & & \\
\gamma & ::= & \varepsilon \mid (b, \gamma) \mid (\overline{b}, \gamma) \mid \langle l_i \to \gamma_i \rangle \mid \beta \mid \mu\beta.\gamma \\
\Theta & ::= & \emptyset \mid \Theta(c :^{A}_{B} \gamma)
\end{array}
$$

Figure 7: Syntax of λ_{MT}

5. MULTI-TIER CALCULUS

The multi-tier calculus, λ_{MT}, is our first calculus with explicit communication instructions. In the previous calculus, λ'_A, the statement $x = \texttt{trans}^{[A \rightsquigarrow B]}\ y$ just states the necessity of a communication between nodes A and B in a declarative way. In contrast, λ_{MT} augments $x = \texttt{trans}^{[A \rightsquigarrow B]}\ y$ with a channel argument and provides primitives that explicitly open and close a communication channel.

Figure 7 defines the syntax of the multi-tier calculus λ_{MT}. The main extension of this calculus with respect to λ_A consists of statements to open a channel (`open`), to transfer data via a connection (`trans`), and close the connection (`close`). In comparison to λ'_A, the `trans` statement obtains a channel parameter and functions receive additional channel parameters $\tilde{c}$. A channel value cannot be bound to a normal variable because the channel changes its type with every communication. Channel variables are treated linearly to simplify the tracking of the change of type.

Further extensions are added already at this time so that the remaining transformation steps can all be expressed without leaving the calculus λ_{MT}. After splitting, an `open` statement yields two statements that have to open channels of matching type. The calculus has an expression $\nu p.e$ that introduces a fresh port name and an `openP` statement that opens a channel with a type prescribed by the port p. Opening channels with the same port ensures that the channels' session types match even if they occur in different places in the program. Freshness of port names ensures that each port value occurs at most once during the run of a program. As usual, introduction of fresh names is commutative, that is, we consider expressions modulo the smallest compatible equivalence relation $\equiv$ containing

$$
\begin{array}{lcl}
\nu p.e & \equiv & e \quad \text{if } p \notin fv(e) \\
\nu p.\nu p'.e & \equiv & \nu p'.\nu p.e.
\end{array}
$$

Finally, the statement, $c\{\ell\}$, applies a channel to a statically known label. Like a type application, it has no operational effect. Intuitively, $c\{\ell\}$ selects one of several labeled types for the channel c: if c has session type $\langle \ell \to \gamma, \ell_1 \to \gamma_1, \ldots \rangle$, then its type changes to γ after the channel application. Section 5.2 explains more about its role in typing.

The translation $\mathcal{T}[\![\cdot]\!]$ from λ'_A to λ_{MT} is straightforward.

$$
\boxed{\Theta, \Gamma \vdash'' e\,!\,N} \qquad \emptyset, \Gamma \vdash'' \texttt{halt}\,!\,\emptyset
$$

$$
\frac{\Theta, \Gamma \vdash'' d \Rightarrow \Theta', \Gamma'\,!\,N_d \qquad \Theta', \Gamma' \vdash'' e\,!\,N_e}{\Theta, \Gamma \vdash'' \texttt{let}\ d\ \texttt{in}\ e\,!\,N_d \cup N_e}
$$

$$
\frac{\Gamma \vdash''_a x : (b, N) \qquad N_1, N_2 \subseteq N \qquad \Theta, \Gamma \vdash'' e_1\,!\,N_1 \qquad \Theta, \Gamma \vdash'' e_2\,!\,N_2}{\Theta, \Gamma \vdash'' \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2\,!\,N_1 \cup N_2}
$$

$$
\frac{\Gamma \vdash''_a x : [\tilde{\gamma}]\tilde{\tau} \xrightarrow{N} 0 \qquad \Gamma \vdash''_a \tilde{z} : \tilde{\tau} \qquad \Theta = (\tilde{c} : \tilde{\gamma})}{\Theta, \Gamma \vdash'' x\,[\tilde{c}]\,(\tilde{z})\,!\,N}
$$

$$
\frac{\Theta(c :^{A}_{B} \gamma), \Gamma \vdash'' e\,!\,N}{\Theta, \Gamma \vdash'' \nu c^{[A \rightsquigarrow B]}.e\,!\,N}
$$

$$
\frac{\Theta, \Gamma(p : \texttt{Port}\ \gamma) \vdash'' e\,!\,N}{\Theta, \Gamma \vdash'' \nu p.e\,!\,N}
$$

Figure 9: Typing rules for target (λ_{MT}) expressions

It replaces each occurrence of a statement

$$
\texttt{let}\ x = \texttt{trans}^{[A \rightsquigarrow B]}\ y\ \texttt{in}
$$

by a sequence of statements that opens an explicit channel between locations A and B, transmits the current value of y from A to B, binds the value to x, and closes both ends of the connection:

$$
\begin{array}{l}
\texttt{let}\ c = \texttt{open}^{[A \rightsquigarrow B]}\ \texttt{in} \\
\texttt{let}\ x = \texttt{trans}^{[A \rightsquigarrow B]}\ c(y)\ \texttt{in} \\
\texttt{let}\ \texttt{close}^{[A \rightsquigarrow B]}\ (c)\ \texttt{in}
\end{array}
$$

5.1 Dynamic Semantics

The definition of the dynamic semantics requires the extension of the syntax with the same values as for λ'_Ain Section 4.1. Additionally, a new binder is needed to model an open communication channel. The expression

$$
e \quad ::= \quad \ldots \mid \nu c^{[A \rightsquigarrow B]}.e
$$

introduces a fresh linear name, $c^{[A \rightsquigarrow B]}$, for a channel between locations A and B.

Figure 8 contains the reduction rules, $\overset{w}{\rightharpoonup}_{t2}$, that handle the new communication statements, together with evaluation contexts, E, that extend the reduction rules under binding constructs.

$$
E ::= [\ ] \mid \nu p.E \mid \nu c^{[A \rightsquigarrow B]}.E
$$

Opening a channel introduces a fresh channel name. The transmission of a value is only possible through a channel connecting the appropriate locations. Closing a channel amounts to the removal of the channel binder for $c^{[A \rightsquigarrow B]}$. The application of a channel to a label has no operational effect.

For λ_{MT}, we use three notions of reductions: $\overset{w}{\rightharpoonup}_{MT,pure}$ is the E-compatible closure of $\overset{w}{\rightharpoonup}_{\text{core}} \cup \overset{w}{\rightharpoonup}_{t2}$, $\overset{w}{\rightharpoonup}_{MT}$ is the E-compatible closure of $\overset{w}{\rightharpoonup}_{\text{imp}} \cup \overset{w}{\rightharpoonup}_{t2}$, and $\overset{w}{\rightharpoonup}_{MT(\overline{A})}$ is the E-compatible closure of $\overset{w}{\rightharpoonup}_{\text{core}} \cup \overset{w}{\rightharpoonup}_{t2} \bigcup_{\forall B.B \neq A} \overset{w}{\rightharpoonup}_{\text{op}^B}$.

5.2 Static Semantics

The judgments of the static semantics of λ_{MT} extend the judgments of λ'_A for expressions and statements by a new type linear environment Θ which associates each channel

$$
\begin{array}{rcll}
\texttt{let } c = \texttt{open}^{[A\rightsquigarrow B]} \texttt{ in } e & \xrightarrow{\varepsilon}_{t2} & \nu c^{[A\rightsquigarrow B]}.e[c \mapsto c^{[A\rightsquigarrow B]}] & \\
\texttt{let } c = \texttt{openP}^{[A\rightsquigarrow B]}\ (p) \texttt{ in } e_1 & \xrightarrow{\varepsilon}_{t2} & \nu c^{[A\rightsquigarrow B]}.e[c \mapsto c^{[A\rightsquigarrow B]}] & \\
\texttt{let } x = \texttt{trans}^{[A\rightsquigarrow B]}\ c^{[A\rightsquigarrow B]}(\texttt{val}(i; N)) \texttt{ in } e & \xrightarrow{\varepsilon}_{t2} & e[x \mapsto \texttt{val}(i; N \cup \{B\})] & \textbf{if } A \in N \\
\nu c^{[A\rightsquigarrow B]}.\texttt{let close}^{[A\rightsquigarrow B]}\ (c^{[A\rightsquigarrow B]}) \texttt{ in } e & \xrightarrow{\varepsilon}_{t2} & e & \\
\texttt{let } c^{[A\rightsquigarrow B]}\{l\} \texttt{ in } e & \xrightarrow{\varepsilon}_{t2} & e &
\end{array}
$$

Figure 8: Reduction rules for λ_{MT}

$$\boxed{\Theta, \Gamma \vdash'' d \Rightarrow \Theta', \Gamma' \,!\, N}$$

$$\frac{(\forall 1 \le i \le n)\ \Gamma \vdash''_a x_i : (b, N)}{\begin{array}{c}\Theta, \Gamma \vdash'' x = \texttt{pfun}(x_1, \ldots, x_n) \\ \Rightarrow\quad \Theta, \Gamma(x : (b, N)) \,!\, \varepsilon\end{array}}$$

$$\frac{(\forall 1 \le i \le n)\ \Gamma \vdash''_a x_i : (b, \{A\})}{\begin{array}{c}\Theta, \Gamma \vdash'' x = \texttt{op}^A(x_1, \ldots, x_n) \\ \Rightarrow\quad \Theta, \Gamma(x : (b, \{A\})) \,!\, \texttt{op}^A()\end{array}}$$

$$\frac{(\forall j)\ (\tilde{c}_j :^{A_j} \tilde{\gamma}_j), \Gamma(f_i : [\tilde{\gamma}]_i \tilde{\tau}_i \xrightarrow{N_i} 0)_{i=1}^n (\tilde{x}_j : \tilde{\tau}_j) \vdash'' e_j \,!\, N_j}{\begin{array}{c}\Theta, \Gamma \vdash'' \texttt{rec } \{\ f_i[\tilde{c}_i](\tilde{x}_i) = e_i\ \}_{i=1}^n \\ \Rightarrow\quad \Theta, \Gamma(f_i : [\tilde{\gamma}]_i \tilde{\tau}_i \xrightarrow{N_i} 0) \,!\, \varepsilon\end{array}}$$

$$\frac{\Gamma \vdash''_a y : (b, N) \quad A \in N}{\begin{array}{c}\Theta(c :^A_B (b, \gamma)), \Gamma \vdash'' x = \texttt{trans}^{[A\rightsquigarrow B]}\ c(y) \\ \Rightarrow\quad \Theta(c :^A_B \gamma), \Gamma(x : \tau, N \cup \{B\}) \,!\, \varepsilon\end{array}}$$

$$\frac{\Gamma \vdash''_a y : (b, N) \qquad A \in N}{\begin{array}{c}\Theta(c :^B_A (\overline{b}, \gamma)), \Gamma \vdash'' x = \texttt{trans}^{[A\rightsquigarrow B]}\ c(y) \\ \Rightarrow\quad \Theta(c :^B_A \gamma), \Gamma(x : \tau, N \cup \{B\}) \,!\, \varepsilon\end{array}}$$

$$\Theta, \Gamma \vdash'' c = \texttt{open}^{[A\rightsquigarrow B]} \Rightarrow \Theta(c :^A_B \gamma), \Gamma \,!\, \varepsilon$$

$$\frac{\Gamma(p) = \texttt{Port } \gamma}{\Theta, \Gamma \vdash'' c = \texttt{openP}^{[A\rightsquigarrow B]}\ (p) \Rightarrow \Theta(c :^A_B \gamma), \Gamma \,!\, \emptyset}$$

$$\Theta(c :^A_B \varepsilon), \Gamma \vdash'' \texttt{close}^{[A\rightsquigarrow B]}\ (c) \Rightarrow \Theta, \Gamma \,!\, \varepsilon$$

$$\Theta(c :^A_B \langle l_i \to \gamma_i \rangle), \Gamma \vdash'' c\{l_j\} \Rightarrow \Theta(c :^A_B \gamma_j), \Gamma$$

Figure 10: Typing rules for target (λ_{MT}) statements

variable to its *session type* and the pair of locations connected by the channel.

A session type, γ, denotes an ω-regular language that describes the sequence of types that the rest of the session communicates over on the channel. The type ε indicates that no further communication can take place on a channel, (b, γ) sends a base value and continues according to γ, and $(\overline{b}, \gamma)$ receives a value and continues. The type $\langle l_1 \to \gamma_1, l_2 \to \gamma_2, \ldots \rangle$ is a conditional session type guarded by the labels $l_1, l_2, \ldots$ The channel application $c\{l_i\}$ changes the type of c to γ_i. Each application of the recursion operator must be expansive, that is, a well-formed session type does not have subterms of the form $\mu\beta_1 \ldots \mu\beta_n.\beta_1$. Such subterms do not correspond to regular trees.

For example, the type from Section 2

$$(\overline{\mathit{string}}, \mu\beta.(\overline{\mathit{boolean}}, \langle \texttt{true} \to (\overline{\mathit{tuple}}, \beta) \mid \texttt{false} \to \varepsilon \rangle))$$

denotes the language

$$\begin{array}{rl} & \{\overline{\mathit{string}}\ \overline{\mathit{boolean}}\ (\overline{\mathit{tuple}}\ \overline{\mathit{boolean}})^n \mid n \in \mathbf{N}\} \\ \cup & \{\overline{\mathit{string}}\ \overline{\mathit{boolean}}\ (\overline{\mathit{tuple}}\ \overline{\mathit{boolean}})^\omega\} \end{array}$$

Since each communication on a channel changes its session type, the variables in Θ must obey a linear typing discipline.

Figure 9 contains the annotated typing rules for expressions. The rules reflect the previous rule set for λ'_A and impose additional demands on the use of channels. The `halt` expression requires that all channels are closed. The conditional requires that all branches must use the channels in the same way. The rule for let indicates that a statement transforms both environments. A function call pass all channels to the function.

The annotated typing rules for statements appear in Figure 10. Of the original rules, only the rule for function definitions changes significantly. The additional requirement is that a function may only refer to the channels passed as parameters, that is, a function does not have free channel variables. The transmission of a value changes the type of the channel as expected. Closing a channel requires that its session type is ε, whereas opening a channel invents a session type. Applying a channel to a label selects the corresponding alternative in the channel's session type.

The subtyping rules remain unchanged. The argument typing rules change only marginally.

5.3 A Notion of Bisimilarity

To prove the correctness of the presented program transformations we need to state relationships between source programs and their transformed counterparts. To this end, we consider two programs equivalent if the perform the same side-effecting operations in the same order. This notion of equivalence is best captured by a weak bisimulation [12].

The definition of a suitable bisimulation requires that we split our transition rules in two parts: those reductions that do not have a side effect, $\xrightarrow{w}_{\mathcal{R}_1}$, and those that do, $\xrightarrow{w}_{\mathcal{R}_2}$. The corresponding observation relation $\xrightarrow{w}_{\mathcal{R}_1,\mathcal{R}_2}$ makes transitions using $\xrightarrow{w}_{\mathcal{R}_1}$ until we reach a transition with $\xrightarrow{w}_{\mathcal{R}_2}$ keeping only the final observation of $\xrightarrow{w}_{\mathcal{R}_2}$. Formally, the relation $\xrightarrow{w}_{\mathcal{R}_1,\mathcal{R}_2}$ is defined as follows:

$$\texttt{halt} \xrightarrow{\texttt{halt}}_{\mathcal{R}_1,\mathcal{R}_2} \Omega$$

$$\frac{e \xrightarrow{w}_{\mathcal{R}_2} e'}{e \xrightarrow{w}_{\mathcal{R}_1,\mathcal{R}_2} e'} \qquad \frac{e \xrightarrow{w'}_{\mathcal{R}_1} e' \quad e' \xrightarrow{w}_{\mathcal{R}_1,\mathcal{R}_2} e''}{e \xrightarrow{w}_{\mathcal{R}_1,\mathcal{R}_2} e''}$$

The `halt` expression is related to a non-terminating term Ω. If we make one observable step with $\xrightarrow{w}_{\mathcal{R}_2}$, then the related expressions are also in the observation relation. Unobservable steps may be prepended to the observation relation.

Following Milner [10] and Gordon [3], we define weak bisimilarity co-inductively—that is, as greatest fixed point—using the following two functions $[-], \langle - \rangle$ that are parame-

terized with respect to four transition relations:

$$(\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)[\mathcal{S}] \stackrel{\text{def}}{=} \{(e,f) \mid \text{if } e \xrightarrow{w}_{\mathcal{R}_1,\mathcal{R}_2} e' \text{ then } \exists f' \text{ with } f \xrightarrow{w}_{\mathcal{R}_3,\mathcal{R}_4} f' \text{ and } e' \;\mathcal{S}\; f'\}$$

$$(\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)\langle\mathcal{S}\rangle \stackrel{\text{def}}{=} (\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)[\mathcal{S}] \cap (\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)[\mathcal{S}^{\text{op}}]^{\text{op}}$$

$$\approx_{\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4} \stackrel{\text{def}}{=} \text{gfp}S.\langle(\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)[\mathcal{S}]\rangle$$

$(\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)[\mathcal{S}]$ denotes pairs that reach a pair in $\mathcal{S}$ by a simulated common step, $(\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4)\langle\mathcal{S}\rangle$ is the restriction of the simulation in both ways. Weak bisimilarity, $\approx_{\mathcal{R}_1,\mathcal{R}_2,\mathcal{R}_3,\mathcal{R}_4}$, is the greatest relation with these properties, *i.e.*, the greatest fixpoint of $\langle\mathcal{S}\rangle$.

5.4 Technical Results

First, type soundness is established in the usual way from a type preservation result and a progress result.

LEMMA 4 (TYPE PRESERVATION). *If* $\emptyset,\emptyset \vdash'' e \,!\, N$ *and* $e \xrightarrow{w}_{MT} e'$ *then* $\emptyset,\emptyset \vdash'' e' \,!\, N'$ *with* $N' \subseteq N$ *and* $locs(w) \subseteq N$.

LEMMA 5 (PROGRESS). *If* $\emptyset,\emptyset \vdash'' e \,!\, N$ *then either* $e = \texttt{halt}$ *or there exists* e' *such that* $e \xrightarrow{w}_{MT} e'$ *and* $locs(w) \subseteq N$.

The translation $\mathcal{T}[\![\_]\!]$ defined at the beginning of this section preserves typing and results in programs with the same observational behavior.

LEMMA 6. *If* $\Gamma \vdash' e \,!\, N$ *then* $(\exists\Gamma')\ \emptyset,\Gamma' \vdash'' \mathcal{T}[\![e]\!] \,!\, N$.

LEMMA 7. $\mathcal{T}[\![\cdot]\!] \subseteq \approx_{\xrightarrow{w}_{\lambda'_{A,pure}},\xrightarrow{w}_{\text{op}},\xrightarrow{w}_{MT,pure},\xrightarrow{w}_{\text{op}}}$

6. FROM DATAGRAMS TO STREAMS

The transformation $\mathcal{T}[\![]\!]$ from the previous section works correctly, but it produces inefficient programs. Whenever a value from location A is needed at location B, the program sends a datagram: it opens a channel from A to B, sends the value, and closes the channel, again. The inefficiency lies in the cost of connection establishment. Since this cost is much higher than the cost of transmitting one value, it would be better if the cost for connection establishment were amortized across as many value transmissions as possible.

To avoid repeated connection establishment, we define a set of transformation rules on λ_{MT} terms that seek to join `close` and `open` statements for a channel with the goal of reusing the channel for multiple communications. Essentially, the transformation turns a sequence of datagram transmissions between two hosts into a stream connection.

Figure 11 specifies the transformation in terms of a relation $\longrightarrow_{ES}$. The strategy for applying the transformation rules is to float each `open` statement upwards in a list of statements until one of the following holds.

1. The `open` meets a `close` with matching (or reversed) locations. In this case, the two channels are joined together and both statements are eliminated.

2. The `open` reaches the beginning of a function body. If the function is recursive, then the `open` is split off in a separate non-recursive wrapper function. Then, the standard inlining transformation can transport the `open` to all call sites of the function [14]. The transformation stops at functions that are not inlineable (*e.g.*, certain toplevel functions).

3. The `open` reaches the top of a branch of a conditional. There are two possibilities. If the other branch has a matching `open`, then the transformation rule first inserts a channel application $c\{l\}$ with label ℓ_t in the true-branch and ℓ_f in the false-branch. If γ_t and γ_f are the original types of the channels in the true- and the false-branch, then their type becomes $\langle \ell_t \to \gamma_t, \ell_f \to \gamma_f\rangle$ so that the channels can be joined and hoisted in front of the conditional.

   If no matching `open` is available in the other branch, then a transformation rule may introduce a new channel between the source and destination host that is opened and immediately closed. Then joining takes place as before.

Typing is preserved under the compatible closure $\Rightarrow_{ES}$ of the relation $\longrightarrow_{ES}$. The transformation with $\Rightarrow_{ES}$ leads to weakly bisimilar programs.

LEMMA 8. *If* $\Theta,\Gamma \vdash e \,!\, N$ *and* $e \Rightarrow_{ES} e'$ *then there exists* Θ' *such that* $\Theta',\Gamma \vdash e' \,!\, N$.

LEMMA 9. $\Rightarrow_{ES} \subseteq \approx_{\xrightarrow{w}_{MT,pure},\xrightarrow{w}_{\text{op}},\xrightarrow{w}_{MT,pure},\xrightarrow{w}_{\text{op}}}$

7. SPLITTING TRANSFORMATION

Given a type derivation for a λ_{MT} program, the splitting transformation extracts for each location a program slice such that running all slices in parallel on their respective locations is equivalent to running the original program.

The transformation proceeds in three steps. The first step introduces global port names for each connection. Port names serve as globally visible points of contact between the processes generated in step number three. The second step pools together all introductions of port names at the beginning of the program. The third step extracts the slices from the type derivation where each slice contains a separate process for each location.

7.1 Port Introduction

The first step of the splitting transformation, the introduction of ports, $\mathcal{P}[\![\cdot]\!]$, replaces each occurrence of

$$\texttt{let } c = \texttt{open}^{[A\rightsquigarrow B]} \texttt{ in } \ldots$$

by a `let` expression with `openP` header that refers to an explicit port name and is surrounded by a ν abstraction introducing precisely that port name:

$$\nu p.(\texttt{let } c = \texttt{openP}^{[A\rightsquigarrow B]}\ (p) \texttt{ in } \ldots)$$

The translation $\mathcal{P}[\![\_]\!]$ preserves typing. The translation produces weakly bisimilar programs.

LEMMA 10. *If* $\Theta,\Gamma \vdash'' e \,!\, N$ *then* $\Theta,\Gamma \vdash'' \mathcal{P}[\![e]\!] \,!\, N$.

LEMMA 11. $\mathcal{P}[\![\cdot]\!] \subseteq \approx_{\xrightarrow{w}_{MT,pure},\xrightarrow{w}_{\text{op}},\xrightarrow{w}_{MT,pure},\xrightarrow{w}_{\text{op}}}$

7.2 Port Floating

The next step moves all port binders to the beginning of the program. Figure 12 specifies the transformation in terms of a relation $\longrightarrow_{PB}$. The second rule lifts a port binder out of an arbitrary function definition in the block.

Typing is preserved under the compatible closure $\Rightarrow_{PB}$ of the relation $\longrightarrow_{PB}$. Transforming with $\Rightarrow_{PB}$ leads to weakly bisimilar programs.

$$
\begin{array}{rcll}
\texttt{let}\ d\ \texttt{in let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ e & \longrightarrow_{ES} & \texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in let}\ d\ \texttt{in}\ e & \textbf{if}\ x \notin \mathit{var}(d) \\
\texttt{let}\ c = \texttt{open}^{[B\rightsquigarrow A]}\ \texttt{in}\ e & \longrightarrow_{ES} & \texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ e & \\
\texttt{let close}^{[A\rightsquigarrow B]}\ (c)\ \texttt{in let}\ c' = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ e & \longrightarrow_{ES} & e[c' \mapsto c] & \\
\texttt{let rec}\ \{\ f[\tilde{c}](\tilde{x}) = \texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ e; \\ \quad r \\ \}\ \texttt{in}\ e' & \longrightarrow_{ES} & \texttt{let rec}\ \{\ f'[\tilde{c}, c](\tilde{x}) = e; \\ \quad f[\tilde{c}](\tilde{x}) = \texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ f'\,[\tilde{c}, c]\,(\tilde{x}); \\ \quad r \\ \}\ \texttt{in}\ e' & \\
\texttt{if}\ x'\ \texttt{then}\ (\texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ e_1) \\ \quad \texttt{else}\ (\texttt{let}\ c' = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in}\ e_2) & \longrightarrow_{ES} & \texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in} \\ \texttt{if}\ x'\ \texttt{then let}\ c\{\ell_t\}\ \texttt{in}\ e_1 \\ \quad \texttt{else let}\ c\{\ell_f\}\ \texttt{in}\ e_2[c' \mapsto c] & \textbf{if}\ \ell_t \neq \ell_f \\
e & \longrightarrow_{ES} & \texttt{let}\ c = \texttt{open}^{[A\rightsquigarrow B]}\ \texttt{in let close}^{[A\rightsquigarrow B]}\ (c)\ \texttt{in}\ e &
\end{array}
$$

Figure 11: Extending the scope of a channel (λ_{MT})

$$
\begin{array}{rcll}
\texttt{let}\ d\ \texttt{in}\ \nu p.e & \longrightarrow_{PB} & \nu p.\texttt{let}\ d\ \texttt{in}\ e & \textbf{if}\ p \notin \mathit{var}(d) \\
\texttt{let rec}\ \{\, f[\tilde{c}](\tilde{x}) = \nu p.e; r \,\}\ \texttt{in}\ e' & \longrightarrow_{PB} & \nu p.\texttt{let rec}\ \{\, f[\tilde{c}](\tilde{x}) = e; r \,\}\ \texttt{in}\ e' & \textbf{if}\ p \notin \mathit{var}(r) \cup \{f, \tilde{x}\} \cup \mathit{fv}(e')
\end{array}
$$

Figure 12: Floating of port binders

LEMMA 12. *If* $\Theta, \Gamma \vdash e\,!\,N$ *and* $e \Rightarrow_{PB} e'$ *then* $\Theta, \Gamma \vdash e'\,!\,N$.

LEMMA 13. $\Rightarrow_{PB} \ \subseteq\ \approx_{\overset{w}{\rightarrow}_{MT,pure}, \overset{w}{\rightarrow}_{\mathrm{op}}, \overset{w}{\rightarrow}_{MT,pure}, \overset{w}{\rightarrow}_{\mathrm{op}}}$

7.3 λ_{MT} Without Magic

As a last preparation of the splitting transformation, we replace the multi-location communication operations in λ_{MT} by traditional communication operations and add an operator for concurrent execution. For example, the $x = \texttt{trans}^{[A\rightsquigarrow B]}\ c(y)$ statement of λ_{MT} specifies sending of y at location A **and** receiving of x at location B (via channel c) at the same time. Also, the $\texttt{open}^{[A\rightsquigarrow B]}$ and $\texttt{close}^{[A\rightsquigarrow B]}\ (c)$ statements perform operations at two locations. Here is the additional syntax of the final calculus λ_{MTC}.

$$
\begin{array}{l}
d ::= \ldots \mid \texttt{close}^{A}\ (c) \mid c = \texttt{listen}^{A}(p) \mid c = \texttt{connect}^{B}(p) \\
\qquad \mid \texttt{send}^{A}\ \ c(x) \mid x = \texttt{recv}^{B}\ (c) \\
e ::= \ldots \mid e \parallel e
\end{array}
$$

The expression $e_1 \parallel e_2$ specifies the interleaved execution of e_1 and e_2. The other statements are the one-sided versions of the previous communication statements. Instead of an unspecific `open` statement for establishing a channel between two locations, there are now `listen` and `connect` to create a server end and a client end of a channel for a specific port. In the same manner, there are separate communication operations to `send` an outbound message and to `recv` an inbound message over an established channel.

7.4 Dynamic Semantics

Again, the intermediate states need additional syntax: The $\nu c.e$ expression introduces a *pair* of fresh *linear* names, c and $\overline{c}$, that model the two ends of a channel.

There are two notions of reduction for λ_{MTC}: $\overset{w}{\rightarrow}_{MTC,pure}$ is the E-compatible closure of $\overset{w}{\rightarrow}_{\mathrm{core}} \cup \overset{w}{\rightarrow}_{t3}$, and $\overset{w}{\rightarrow}_{MTC}$ is the E-compatible closure of $\overset{w}{\rightarrow}_{\mathrm{imp}} \cup \overset{w}{\rightarrow}_{t3}$ (cf. Figure 13). Evaluation contexts are defined by $E ::= [\,] \mid E \parallel e \mid \nu p.E \mid \nu c.E$. Both relations consider expressions modulo the smallest compatible equivalence relation $\equiv$ satisfying (for both kinds of ν).

$$
\begin{array}{lcll}
\nu x.e & \equiv & e & \text{if } x \notin \mathit{fv}(e) \\
\nu x.\nu y.e & \equiv & \nu y.\nu x.e & \\
e \parallel e' & \equiv & e' \parallel e & \\
e \parallel (e' \parallel e'') & \equiv & (e \parallel e') \parallel e'' & \\
(\nu x.e) \parallel e' & \equiv & \nu x.(e \parallel e') & \text{if } x \notin \mathit{fv}(e').
\end{array}
$$

The reduction rules in Fig. 13 work as follows. The pairing of a `listen` command and a `connect` command for the same port results in a channel abstraction where the channel names are replaced by the paired names c and $\overline{c}$. A `send`/`recv` pair for the same channel c results in transferring the value v from one end to the other. Closing both ends of a channel amounts dropping the channel entirely. The application of a channel to a label is a synchronization construct. It requires the same label application at the other end of the channel to proceed.

For λ_{MTC}, we use $\overset{w}{\rightarrow}_{MTC(\overline{A})}$, the E-compatible closure of $\overset{w}{\rightarrow}_{\mathrm{core}} \cup \overset{w}{\rightarrow}_{t3} \bigcup_{\forall B.B \neq A} \overset{w}{\rightarrow}_{\mathrm{op}^B}$, as the notion of reduction.

7.5 Static Semantics

Figure 14 contain the typing rules for the new statements and expression. Each port created by $\nu p.e$ has a fixed session type associated with it. Since a port is globally available, its type, `Port` γ, does not carry a location set. The idea is that $c = \texttt{listen}^{A}(p)$ binds c to the server end of a channel at location A. The channel inherits its session type γ from port p. $c = \texttt{connect}^{A}(p)$ creates the client end. Since sending and receiving of data is exactly reversed on the client, the session type is *mirrored*—indicated by overlining (cf. Figure 15)—before it is assigned to the new client end. The `send` and `recv` operations peel off one communication event from the channel type; `send` an outbound event, and `recv` an inbound event. The revised `close` operation only closes one end of a channel.

The rule for concurrent execution splits the linear channel environment into two disjoint parts, one for each subprocess, as indicated by the + operator. The value environment is

$$\begin{array}{rcl}
\texttt{let } c_1 = \texttt{listen}^A(p) \texttt{ in } e_1 \parallel \texttt{let } c_2 = \texttt{connect}^B(p) \texttt{ in } e_2 & \xrightarrow{\varepsilon}_{t3} & \nu c.(e_1[c_1 \mapsto c] \parallel e_2[c_2 \mapsto \overline{c}]) \\
\texttt{let send}^A\ c(v) \texttt{ in } e_1 \parallel \texttt{let } x = \texttt{recv}^B(\overline{c}) \texttt{ in } e_2 & \xrightarrow{\varepsilon}_{t3} & e_1 \parallel e_2[x \mapsto v] \\
\nu c.(\texttt{let close}^A\ (c) \texttt{ in } e_1 \parallel \texttt{let close}^B\ (\overline{c}) \texttt{ in } e_2) & \xrightarrow{\varepsilon}_{t3} & e_1 \parallel e_2 \\
\texttt{let } c\{l\} \texttt{ in } e_1 \parallel \texttt{let } \overline{c}\{l\} \texttt{ in } e_2 & \xrightarrow{\varepsilon}_{t3} & e_1 \parallel e_2
\end{array}$$

Figure 13: Reduction rules for λ_{MTC}

$\boxed{\Theta, \Gamma \vdash'' e \,!\, N}$

$$\frac{\Theta_1, \Gamma \vdash'' e_1 \,!\, N_1 \qquad \Theta_2, \Gamma \vdash'' e_2 \,!\, N_2}{\Theta_1 + \Theta_2, \Gamma \vdash'' e_1 \parallel e_2 \,!\, N_1 \cup N_2}$$

$\boxed{\Theta, \Gamma \vdash'' d \Rightarrow \Theta', \Gamma' \,!\, N}$

$$\frac{\Gamma(p) = \texttt{Port } \gamma}{\Theta, \Gamma \vdash'' c = \texttt{listen}^A(p) \Rightarrow \Theta(c :^A \gamma), \Gamma \,!\, \emptyset}$$

$$\frac{\Gamma(p) = \texttt{Port } \gamma}{\Theta, \Gamma \vdash'' c = \texttt{connect}^A(p) \Rightarrow \Theta(c :^A \overline{\gamma}), \Gamma \,!\, \emptyset}$$

$$\frac{\Gamma \vdash''_a x : (b, N) \qquad A \in N}{\Theta(c :^A (b, \gamma)), \Gamma \vdash'' \texttt{send}^A\ c(x) \Rightarrow \Theta(c :^A \gamma), \Gamma \,!\, \emptyset}$$

$$\Theta(c :^A (\overline{b}, \gamma)), \Gamma \vdash'' x = \texttt{recv}^A(c) \Rightarrow \Theta(c :^A \gamma), \Gamma(x : b) \,!\, \emptyset$$

$$\Theta(c :^A \varepsilon), \Gamma \vdash'' \texttt{close}^A\ (c) \Rightarrow \Theta, \Gamma \,!\, \emptyset$$

Figure 14: Typing for extended target syntax, λ_{MTC}

$$\begin{array}{lcllcl}
\overline{\langle l_i \to \gamma_i \rangle} & = & \langle l_i \to \overline{\gamma_i} \rangle & \overline{\varepsilon} & = & \varepsilon \\
\overline{\mu\beta.\gamma} & = & \mu\beta.\overline{\gamma} & \overline{\beta} & = & \beta \\
\overline{\upsilon, \gamma} & = & \overline{\upsilon}, \overline{\gamma} & \overline{\overline{\upsilon}, \gamma} & = & \upsilon, \overline{\gamma}
\end{array}$$

Figure 15: Mirroring of channel types

copied to both subprocesses and their effect is gathered.

7.6 Slice Extraction

The final step starts with a type derivation for $\emptyset, \emptyset \vdash'' \nu\tilde{p}.e \,!\, N$. The transformation $\mathcal{T}^e_N\,[\![\nu\tilde{p}.e]\!]$ in Figure 16 skips over the leading port binders, extracts for each $A_i \in N$ the slice of operationsfor location A_i, and puts them in parallel. The resulting program has the form $\nu\tilde{p}.(e'_1 \parallel \ldots \parallel e'_m)$. Strictly speaking, the transformation requires a type derivation as input. To avoid this clutter, the input term carries annotations. The annotation $@N$ provides the inferred effect of an expression or statement. When handling a channel, the transformation needs to know the source and target locations of the channel. Again, the annotation $@\{B, C\}$ provides this information. Both pieces of information are present in a type derivation.

LEMMA 14. *Suppose that* $\emptyset, \emptyset \vdash'' \nu\tilde{p}.e \,!\, N$. *Then* $\emptyset, \emptyset \vdash'' \mathcal{T}^e_N\,[\![\nu\tilde{p}.e]\!] \,!\, N$.

The transformed program is weakly bisimilar to the original one if we consider effects on each location A separately.

LEMMA 15. *Let* $\emptyset, \emptyset \vdash'' \nu\tilde{p}.e!N$. *For each location* $A \in N$, *it holds that* $\mathcal{T}^e_N\,[\![\cdot]\!] \subseteq \approx_{\xrightarrow{w}_{MT(\overline{A})}, \xrightarrow{w}_{\mathrm{op}^A}, \xrightarrow{w}_{MTC(\overline{A})}, \xrightarrow{w}_{\mathrm{op}^A}}$.

8. RELATED WORK

The splitting transformation is closely related to program slicing [17]. Recent advances in slicing deal with concurrent programs [11, 8], however, while slicing does not introduce new operations, our transformation maps a sequential program into a multi-threaded program. Another difference is that slicing is usually driven by the program dependency graph, whereas our transformation is driven by location analysis which includes dependency information through the typing rule for the conditional.

Secure program partitioning (SPP) [19] is a closely related transformation that maps a sequential program to a distributed program. However, the goals of SPP are quite different. SPP starts from a program with confidentiality and integrity annotations for functions, data, and a number of hosts. From these annotations, SPP partitions the program such that each host runs that part of the program for which its credentials are appropriate. The partition further ensures that the host only receives data up to its confidentiality level and that data from the host is only trusted up to its integrity level. In an extension of that work [20], the authors use replication to increase the scope of SPP.

Binding-time analysis [7, 5] can be seen as a special case of the location analysis. For example, the `TRANS` construct is closely related to lifting. However, a binding-time analysis distinguishes two modes of computation (static and dynamic) whereas a location analysis for n locations distinguishes 2^n different modes.

Our calculus may be viewed as an instance of the capability calculus [18] restricted to simple types and specialized to particular resources (channels and locations). This specialization enables us to exploit properties beyond the reach of the capability calculus.

Session types [2] have emerged as an expressive typing discipline for *heterogeneous*, bidirectional communication channels. 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.

Research on the parallel implementation of functional languages is concerned with the automatic detection of implicit parallelism and the speculative evaluation of expressions, *e.g.*, [13, 4]. Our analysis does not detect parallelism but determines independent slices of programs with explicit communication interfaces. There is no speculative evaluation in our transformed programs, although the splitting transformation (guided by the location analysis) may introduce redundancy by performing location independent operations in more than one location simultaneously.

9. CONCLUSIONS AND FUTURE WORK

We have presented a location analysis that enables splitting of an application into slices that execute independently

Toplevel Expressions

$\mathcal{T}^e_N[\![\nu p.e]\!] = \nu p.\mathcal{T}^e_N[\![e]\!]$

$\mathcal{T}^e_{\{A_1,\ldots,A_m\}}[\![e]\!] = \mathcal{T}^e_{A_1}[\![e]\!] \parallel \ldots \parallel \mathcal{T}^e_{A_m}[\![e]\!]$ **if** $e \not\equiv \nu p.e'$

Expressions

$\mathcal{T}^e_A[\![\texttt{halt}]\!] = \texttt{halt}$

$\mathcal{T}^e_A[\![\texttt{let}\ d\ \texttt{in}\ e]\!] = \texttt{let}\ \mathcal{T}^d_A[\![d]\!]\ \texttt{in}\ \mathcal{T}^e_A[\![e]\!]$

$$\mathcal{T}^e_A[\![\texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2@N]\!] = \begin{cases} \texttt{if}\ x\ \texttt{then}\ \mathcal{T}^e_A[\![e_1]\!]\ \texttt{else}\ \mathcal{T}^e_A[\![e_2]\!] & A \in N \\ \texttt{halt} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^e_A[\![x\,[\tilde{c}]\,(\tilde{z})@N]\!] = \begin{cases} (x)\,[\mathcal{T}^t_A[\![\tilde{c}]\!]\,\mathcal{T}^t_A[\![\tilde{z}]\!] & A \in N \\ \texttt{halt} & \text{otherwise} \end{cases}$$

Statements

$$\mathcal{T}^d_A[\![x = \texttt{pfun}(\tilde{x})@N]\!] = \begin{cases} x = \texttt{pfun}(\tilde{x}) & A \in N \\ \texttt{rec}\ \{\ \} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^d_A\left[\!\!\left[x = \texttt{op}^{A'}(\tilde{x})\right]\!\!\right] = \begin{cases} x = \texttt{op}^{A'}(\tilde{x}) & A = A' \\ \texttt{rec}\ \{\ \} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^d_A\left[\!\!\left[s = \texttt{openP}^{[B\rightsquigarrow C]}\ (p)\right]\!\!\right] = \begin{cases} c = \texttt{listen}^A(p) & B = A \\ c = \texttt{connect}^A(p) & C = A \\ \texttt{rec}\ \{\ \} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^d_A\left[\!\!\left[\texttt{close}^{[B\rightsquigarrow C]}\ (c)\right]\!\!\right] = \begin{cases} \texttt{close}^A\ (c) & A = B \vee A = C \\ \texttt{rec}\ \{\ \} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^d_A[\![\texttt{rec}\ \{\ f_i[\tilde{c}_i](\tilde{x}_i) = e_i\ \}^n_{i=1}]\!] = \texttt{rec}\ \{\ f_i[\mathcal{T}^t_A[\![\tilde{c}_i]\!]](\mathcal{T}^t_A[\![\tilde{x}_i]\!]) = \mathcal{T}^e_A[\![e_i]\!]\ \}^n_{i=1}$$

$$\mathcal{T}^d_A\left[\!\!\left[\texttt{trans}^{[B\rightsquigarrow C]}\ c(x)\right]\!\!\right] = \begin{cases} \texttt{send}\ c(x) & A = B \\ x = \texttt{recv}(c) & A = C \\ \texttt{rec}\ \{\ \} & \text{otherwise} \end{cases}$$

$$\mathcal{T}^d_A[\![c\{l_j\}^B_C]\!] = \begin{cases} c\{l_j\} & A = B \vee A = C \\ \texttt{rec}\ \{\ \} & \text{otherwise} \end{cases}$$

Parameter lists

$\mathcal{T}^t_A[\![\epsilon]\!] = ()$

$$\mathcal{T}^t_A[\![\tilde{x}, x@N]\!] = \begin{cases} \mathcal{T}^t_A[\![\tilde{x}]\!], x & A \in N \\ \mathcal{T}^t_A[\![\tilde{x}]\!] & \text{otherwise} \end{cases}$$

Figure 16: Slice extraction in λ_{MTC}

in parallel and that communicate via typed streams. The analysis and the accompanying transformations enable the development of distributed applications in a local setting. Each of the transformation steps is proven correct.

Ongoing work considers the efficient implementation of the location analysis. The challenge is that the analysis must be configurable with respect to location preferences and communication requirements. The present work only gives a specification. Practical experience will show if a polymorphic analysis is required.

In general, the mapping to tiers will be a multi-language issue that requires an additional translation step from the framework's language to the desired target languages. Alternatively, a heterogenous framework might be considered where either the analysis applies directly to different languages.

Acknowledgment

We are indebted to the anonymous reviewers whose numerous and extensive comments helped to improve the presentation significantly.

10. REFERENCES

[1] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the 1993 Conference on Programming Language Design and Implementation*, pages 237–247, Albuquerque, New Mexico, June 1993.

[2] S. Gay and M. Hole. Types and subtypes for client-server interactions. In D. Swierstra, editor, *Proceedings of the 1999 European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 74–90, Amsterdam, The Netherlands, Apr. 1999. Springer-Verlag.

[3] A. Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, 228(1-2):5–47, Oct. 1999.

[4] J. Greiner and G. E. Blelloch. A provably time-efficient parallel implementation of full speculation. *ACM Transactions on Programming Languages and Systems*, 21(2):240–285, 1999.

[5] F. Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [6], pages 448–472.

[6] J. Hughes, editor. *Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, MA, 1991. Springer-Verlag.

[7] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[8] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings 9th European Software Engineering Conference*, pages 178–187. ACM Press, 2003.

[9] X. Leroy. *The Objective Caml system release 3.02, Documentation and user's manual*. INRIA, France, July 2001. `http://pauillac.inria.fr/caml`.

[10] R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[11] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 180–190. ACM Press, 2000.

[12] D. Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, number 104 in Lecture Notes in Computer Science, pages 167–1183. Springer-Verlag, 1981.

[13] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[14] S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In Hughes [6], pages 636–666.

[15] W. R. Stevens. *UNIX Network Programming*. Prentice Hall Software Series, 1990.

[16] Y. M. Tang and P. Jouvelot. Effect systems with subtyping. In W. Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 45–53, La Jolla, CA, USA, June 1995. ACM Press.

[17] F. Tip. A survey of program slicing techniques. *J. Programming Languages*, 3(3):121–189, 1995.

[18] D. Walker, C. Crary, and G. Morrisett. Typed memory management via static capabilities. *ACM Transactions on Programming Languages and Systems*, 22(4):701–771, July 2000.

[19] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, 2002.

[20] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 236. IEEE Computer Society, 2003.