Technical Report

UCAM-CL-TR-557
ISSN 1476-2986

Number 557

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Iota: A concurrent XML scripting language with applications to Home Area Networking

G.M. Bierman, P. Sewell

January 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2003 G.M. Bierman, P. Sewell

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Iota: A concurrent XML scripting language with applications to Home Area Networks

G.M. Bierman P. Sewell
University of Cambridge Computer Laboratory,
J.J. Thomson Avenue, Cambridge, CB3 0FD.
{gmb,pes20}@cl.cam.ac.uk

Abstract

Iota is a small and simple concurrent language that provides native support for functional XML computation and for typed channel-based communication. It has been designed as a domain-specific language to express device behaviour within the context of Home Area Networking.

In this paper we describe Iota, explaining its novel treatment of XML and describing its type system and operational semantics. We give a number of examples including Iota code to program Universal Plug 'n' Play (UPnP) devices.

Contents

1 Introduction

Rapid advances in network and device technologies are accelerating the vision of pervasive networking and ubiquitous computing. One such area is in the home: one can reasonably expect homes in the near future to have some form of local area network (a so-called Home Area Network, or HAN), and future consumer devices to exploit this interconnectivity. Various media sources (e.g. TV receivers, CD players) and sinks (e.g. video displays, amplifier/speaker combinations) will become integrated with phone access, traditional home-automation of lighting and heating, etc. The ability to 'script' the whole house will be key. Indeed, consumer device manufacturers are already formulating proposals for how their devices may communicate and cooperate within such a future home, e.g. the XML-based UPnP [upn00]. Software developers will face new challenges in this environment. Typical applications will involve concurrent scripting, and some form of communication; features that are rather heavyweight in conventional programming languages. In addition, consumer devices will offer only a small software platform, and to reduce device time-to-market it will be important to make code development relatively straightforward.

The AutoHAN project in the Cambridge Computer Laboratory [SGG01, BH01] is investigating various aspects of the architecture, programming and user-interface issues within a future Home Area Network. We have designed and implemented a domain specific language, Iota, to address some of the software engineering issues of developing code in this environment. Some of the problems we have addressed are:

- **Concurrency:** A HAN will consist of a large number of devices, executing concurrently, and communicating often. Thus we need a simple, lightweight, and flexible paradigm for writing highly concurrent scripts.
- **XML:** It is rapidly becoming clear that XML will be the *lingua franca* for communication in future networks. Indeed, the language for device description and communication in the UPnP standard is XML. We need to support native syntax for creation and examination of XML values in our programming language, rather than simply providing library calls for dealing with XML as in Java.
- **Elegance:** A HAN is a critical piece of infrastructure; programmable devices need to be reliable and have easily predictable behaviour. Given that this home setting is unlikely to require the ultra-fast execution of code, the key problem for the software developer is to quickly develop clear and predictable code. Thus we need a simple programming language, with well-defined and clear semantics, and at a high level of abstraction. The domain is intrinsically concurrent, but for writing predictable code it is useful to have a clearly-identifiable functional fragment of the language.
- **Strong typing:** Strong typing is a key programming language feature for constructing reliable software. Straightforward type systems for functional and concurrent languages are well-understood, and there is a developing body of work on typing XML computation. In the HAN setting, however, we can assume little about the structure of XML received from other devices – in particular, there may not be standard DTDs or Schema that it is guaranteed to conform to.

- **Correctness:** Given the importance of the correct behaviour of code executing within the home, there will be increased pressure on software developers to assert and verify claims of program correctness. Automated proofs of certain safety properties may even be required. We need to develop a language that is both small enough to feasibly reason about, and also amenable to various techniques for reasoning about program correctness.

In this paper we describe the main design choices underlying Iota, showing how the problems above have been addressed. We first outline the types and syntax of a core language, in §2. The treatment of XML is discussed in §3, and that of concurrency in §4. We give some examples of Iota programming in §5–7, showing how event primitives can be coded up, how a UPnP device can be controlled, and some processing of XML data obtained by HTTP. Iota has a semantic definition, comprising a type system and an operational semantics, and has been implemented. The definition is given in the Appendix.

2 Language Overview and Core Syntax

The core of Iota is an explicitly-typed language, with types as in Figure 1. It provides several base types (distinguished only in that characters and strings are Unicode, to correspond with XML), tuples and lists. Higher-order functions are supported, with call-by-value semantics, and a fixed collection of ML-style exceptions can be raised and handled. The types MU and content will be introduced in the next section, and T chan and proc in the following section. The definition does not currently include parametric polymorphism but, as we shall see, it does involve subtyping. (In fact, the prototype implementation provides parametric polymorphism also.)

For concreteness we give the full syntax of the core language in Figure 2, in which constants and other terminals are as in Figure 1. Much is standard; the novel aspects are discussed in the subsequent sections.

Types T	::=	bool	Booleans
		int	Integers
		char	Characters
		string	Strings
		unit	Unit
		$T_1 * .. * T_n$	Tuples $n \geq 2$
		T list	Lists
		$T \rightarrow T$	Function space
		exn	Exceptions
		MU	Mark-up
		content	Content (to be marked up)
		T chan	Channel carrying T
		proc	Processes

$\underline{i}$	Integer constant
$\underline{b}$	Boolean constant
$\underline{c}$	Character constant
$\underline{s}$	String constant in quotes, eg “ab”
x	Identifier
ex	Exception constructor
tag	Tag
a	Attribute name

Figure 1: IOTA Types and Terminals

Expressions	e	::=	x	Identifier
			$\underline{i}, \underline{b}, \underline{c}, \underline{s}$	Integer, Boolean, Character, String constant
			$()$	Unit
			$(e_1, .., e_n)$	Tuple $n \geq 2$
			$[]$	Empty List
			$e :: e$	Cons
			`if` e `then` e `else` e	Conditional
			`fn` *match*	Function
			`fr` x *match*	Recursive Function
			$e\ e$	Application
			`let` *dec* `in` e	Local declaration
			$ex\, e$	Exception
			`raise` e	Raise exception
			`try` e `with` *match*	Handle exception(s)
			$\langle te\ aes // \rangle e$	Markup
			0	Empty process
			$e \| \| e$	Parallel composition
			`new` $x : T$ `in` e	New channel declaration
			$e ! e$	Output along a channel
			$e ? e$	Input from a channel
			$e ? * e$	Replicated input

Matches	*match*	::=	$p \Rightarrow e$	
			$p \Rightarrow e \mid match$	

Tag expressions	te	::=	*tag*	Constant Tag
			$\{e\}$	Computed Tag

Attribute expression sequences	*aes*	::=	empty	Empty sequence
			$a = e\ aes$	

Patterns	p	::=	$* : T$	Wildcard
			$x : T$	Variable
			$\underline{i}, \underline{b}, \underline{c}, \underline{s}$	Integer, Boolean, Character, String constant
			$()$	Unit
			$(p_1, .., p_n)$	Tuple $n \geq 2$
			$[]$	Empty List
			$p :: p$	Cons
			$ex\ p$	Exception constructor pattern
			$\langle tp\ aps // \rangle p$	Markup pattern

Tag Patterns	*tp*	::=	$*$	Wildcard tag pattern
			tag	Constant tag pattern
			$\{x\}$	Identifier
Attribute Patterns	*ap*	::=	$*$	Wildcard
			$\underline{s}$	String
			x	Identifier
Attribute Pattern Sequences	*aps*	::=	empty	Empty sequence
			$*$	Wildcard sequence
			$a = ap\ aps$	

Declarations	*dec*	::=	`val` $x = e$	Value

Figure 2: IOTA Core Language Syntax

3 Design: XML

One of the main design questions for Iota is how the XML computation should be typed. There are three options:

1. not at all, i.e. treating all XML values as strings;
2. guaranteeing XML *well-formedness*, i.e. that opening and closing tags match and that elements are appropriately nested; or
3. guaranteeing XML *validity*, i.e. well-formedness together with conformance to a DTD, Schema, or other specification.

The last is most desirable, where feasible, as it will exclude many erroneous programs. A number of programming-language type systems for validity have been developed, notably those of XDuce [HP00] and XMλ [SM00]. Unfortunately in the HAN setting it is unclear whether any single notion of schema will become widespread. There are many options – Lee and Chu [LC00] discuss six schema languages in detail and mention four others in passing. In fact, at the start of the Iota design we could obtain sample descriptions of a device (a UPnP-enabled CD player) only as well-formed fragments of XML based on informal 'templates', rather than any DTD or Schema specification. We therefore chose option (2), designing a type system that ensures well-formedness only. This has three further advantages. Firstly, the type system is considerably simpler that those of [SM00, HP00]. Secondly, it allows computation of XML tags, which would be hard to statically type in a system for validity. Thirdly, it may make it easier to write code which is robust under future (unpredictable!) changes of device descriptions, following the 'ignore options that are not understood' principle that has been so successful in network protocols. This is especially important in the highly dynamic world of home device manufacture, where devices are updated rapidly and there is intense competition between manufacturers – universal agreement would be desirable (perhaps using the WSDL language, developed for specifying web services), but seems most unlikely.

One of our design goals was to allow XML to appear in the code in as close a form to the XML standard as possible, with little syntactic 'noise' (beyond that intrinsic to the standard, of course). This contrasts with work where XML elements are coded up using existing language features, e.g. the Haskell XML embedding of Wallace and Runciman [WR99]. A simple XML element can be written as an Iota value as below

$\langle$`person age` = "3"$\rangle$"Tim"$\langle$/`person`$\rangle$

which is as in the standard except for the quotes of "Tim". These are required to disambiguate between constants and computations – the language allows any (well-typed) expression in the same position, for example

`let val` $x = 4$ `in` $\langle$`person age` = "3"$\rangle(x + 5)\langle$/`person`$\rangle$

which shows also an implicit coercion from int to XML, just as the earlier example coerced a string to XML.[1]

Computations in attribute position are also possible, for example

$$\langle \texttt{person name} = \text{“Berners”} \hat{} \text{“-”} \hat{} \text{“Lee”} / \rangle$$

as are computations in tag position, for example

$$\texttt{let } x = \text{“person”} \texttt{ in } \langle \{x\} \texttt{ age} = \text{“7”} / \rangle$$

though for the latter we use extra syntax (the braces { and }) in the computation case rather than the constant case, as we expect the majority of XML expressions will have constant tags. We do not permit computation of attribute *names*, as this would make it hard to statically ensure well-formedness (to prevent repeated names) and as we believe the need will be rare.

Making this precise, the core language has a single XML expression form

$$\langle te\ aes//\rangle e$$

in which te is a tag expression, aes is a sequence of attribute expressions, and e is an expression whose value is to be marked up. There are straightforward derived forms, which are translated out before typechecking as below, to provide the usual outfix and non-fix syntax.

$$\begin{array}{lcll} \langle tag\ aes\rangle e\langle /tag\rangle & \mapsto & \langle tag\ aes//\rangle e & \text{(derived)} \\ \langle te\ aes/\rangle & \mapsto & \langle te\ aes//\rangle [] & \text{(derived)} \end{array}$$

We introduce two types, MU, of XML elements, and content, broadly of values that can be marked-up. For convenience we allow a number of types to be candidates for marking up, defining a subtyping relation with the axioms

$$\mathsf{bool} <: \mathsf{content} \quad \mathsf{int} <: \mathsf{content} \quad \mathsf{char} <: \mathsf{content} \quad \mathsf{string} <: \mathsf{content} \quad \mathsf{MU} <: \mathsf{content}$$

together with the usual rules for functions, tuples, lists, reflexivity and transitivity. To type a markup expression the tag and attribute expressions must be ok (the rules for which are in the Appendix) and the body e must be a content list.

$$\frac{E \vdash te\ \mathsf{ok} \quad E \vdash aes\ \mathsf{ok} \quad E \vdash e : \mathsf{content\ list}}{E \vdash \langle te\ aes//\rangle e : \mathsf{MU}}$$

[1] An alternative would be to allow strings to appear without quotes, and require all program identifiers to be prefixed, e.g. with a dollar sign. We felt that this would be too burdensome for the programmer.

The XML element

```
<A><B>Hello</B> <B>World</B> <F>Hello</F> <F>Universe</F></A>
```

is thus represented in core IOTA as

$\langle \texttt{A}//\rangle\ [\langle \texttt{B}//\rangle \text{“Hello”}, \langle \texttt{B}//\rangle \text{“World”}, \langle \texttt{F}//\rangle \text{“Hello”}, \langle \texttt{F}//\rangle \text{“Universe”}]$

where the body of the `A` tag is a bracket-and-comma delimited list of content.

To eliminate the clutter of these delimiters, and (more importantly) to allow the programmer to write expressions that look like XML elements, we allow certain space-separated sequences of expressions inside a pair of tags:

$\langle tag\ aes\rangle e_1\ ..\ e_n\langle /tag\rangle$

allowing the above to be written as

$\langle \texttt{A}\rangle\ \langle \texttt{B}\rangle \text{“Hello”} \langle /\texttt{B}\rangle\ \langle \texttt{B}\rangle \text{“World”} \langle /\texttt{B}\rangle\ \langle \texttt{F}\rangle \text{“Hello”} \langle /\texttt{F}\rangle\ \langle \texttt{F}\rangle \text{“Universe”} \langle /\texttt{F}\rangle\ \langle /\texttt{A}\rangle$

Indeed, the first simple examples above already made use of this form to omit brackets. Its meaning is not straightforward, however. Consider the IOTA code fragment $\langle \texttt{A}\rangle e\ e' \langle /\texttt{A}\rangle$. The question is how to interpret $e\ e'$? It is ambiguous as it stands: it could be a function application (with $e : T \to \mathsf{content}$ and $e' : T$), in which case the fragment should be regarded as $\langle \texttt{A}\rangle[(e\ e')]\langle /\texttt{A}\rangle$, or a juxtaposition of XML elements, in which case the fragment should be regarded as $\langle \texttt{A}\rangle[e, e']\langle /\texttt{A}\rangle$. There is even a third possibility, if $e : T \to \mathsf{content\ list}$ and $e' : T$, in which case the fragment should be regarded as simply $\langle \texttt{A}\rangle(e\ e')\langle /\texttt{A}\rangle$. Thus *type-based disambiguation* is required to make sense of such expressions. In this paper we do not specify exactly how this is done – the implementation uses an algorithm that seems to work well in practice; a formal description of its properties remains for future work. Note that in longer sequences one must deal also with the fact that cons and function application associate on opposite sides.

We should emphasise that the use of type-based disambiguation is an experimental design choice. Our aim was to allow the XML values within IOTA code to be as close as possible to the actual concrete syntax of XML. In practice, type-based disambiguation does not seem to cause much confusion for programmers, but clearly much more experience is needed. It remains to be seen whether programmers can always easily disambiguate their code, or whether we need to change the IOTA syntax to force disambiguation. (The latter design choice has been taken by the designers of XQuery, who use two different braces to distinguish between XML values and expressions.)

Note. There does appear to be a genuine interaction between XML parsing and strong typing. In the latest version of XQuery (August 2002), it is stated that the XML fragment `<sizes>1 2 3</sizes>` is parsed as `<sizes>"1 2 3"</sizes>` (using IOTA syntax).

However during type-checking (called "Schema validation") it could be re-parsed as, for example, `<sizes>[1,2,3]</sizes>` (again using IOTA syntax). Thus matching a value against a type can change its syntactic structure.

IOTA supports the definition of functions by pattern-matching, much as in ML. The forms of core patterns are shown in Figure 2; they roughly match the expression forms, so for XML we have

$$\langle tp\ aps//\rangle p$$

where *tp* is a tag pattern and *aps* is a sequence of attribute patterns. The latter may end in a wildcard, allowing unknown attributes to be discarded. Attribute matching is unordered. To this are added derived forms

$$\begin{array}{lcll} \langle tag\ aps\rangle p\langle/tag\rangle & \mapsto & \langle tag\ aps//\rangle p & \text{(derived)} \\ \langle tp\ aps/\rangle & \mapsto & \langle tp\ aps//\rangle[] & \text{(derived)} \end{array}$$

and a form that requires type-based disambiguation:

$$\langle tp\ aps\rangle p_1\ p_2\ ...\ p_n\langle/tp\rangle$$

The notion of subtyping in IOTA means that we can emulate a limited form of typecase (in the sense of Abadi et al. [ACPP91]): a form of choice operator where the choice is determined by the *type* of the argument, rather than its value. For example, consider the following code.

$$\begin{array}{l} \texttt{fn}\ x : \mathsf{string} \Rightarrow x \\ \mid x : \mathsf{char} \Rightarrow x \\ \mid x : \mathsf{int} \Rightarrow x + 1 \\ \mid x : \mathsf{content} \Rightarrow x \end{array}$$

This function (of type content → content) acts like the identity function on character and string values, but the increment function for integers.

4 Design: Concurrency

For concurrency and communication we take primitive asynchronous message-passing and parallel composition, based on the π-calculus [MPW92]. Experience with the PICT [PT00] and NOMADIC PICT [SWP99] programming languages shows that this is a lightweight but expressive choice in which many idioms can be coded up, including multi-cast messages, RPCs, locks, and simple objects.

We take a type proc of process expressions and allow parallel composition $e||e'$ of expressions of type proc. The empty process is written 0. Channel names can be created using

the `new` expression. They have types T `chan`, for channels carrying values of type T. The output process (written $e!e'$) takes two arguments: the first e specifying the channel to use (often this will just be a channel name rather than some more complex expression); the second e' gives the value to be sent. Note that there is no continuation after an output – the model is of *asynchronous* communication. The input process (written as $e?e'$) again takes two arguments, the first specifying a channel name and the second being a function which is applied to the received value. For example, consider the following IOTA code.

`new` x : `string chan in` $(x!$"hi"$)||(x?$`fn` y : `string` $\Rightarrow$ `Iota.err`$!(y\hat{}$" there"$))$

This creates a new channel, x, down which the left process sends the string "hi". This string is then read by the other process and is concatenated with another string " there", and thence sent to the built-in channel `Iota.err`. This channel echoes its input to the screen (in this case the string "hi there"). We also provide a repeated input operation (written as infix $e?* e'$). To remove the annoying occurences of the keyword `fn` in the input operations, we extend the core language with the derived form $e?* match$ for the slightly more verbose $e?$`fn` $match$.

There are many language-design choices in how functional and concurrent computation can be integrated, both in type system and operational semantics. Several have been explored in the literature (e.g. in CML, Facile, and JoCaml, among others). We will not discuss the whole design space here, but note only that in IOTA the functional and process parts are layered by the type system; functional reduction cannot spawn new processes. This is an experimental choice – to encourage the writing of robust code we want a clearly-identifiable fragment of the language which is guaranteed not to have communication side-effects (and so no problems with deadlock, etc). It also makes for a simpler semantics. Only experience can show if the consequent loss of expressiveness is tolerable. Communication of higher-order values (including parameterised processes) is included.

Raised exceptions propagate up through functional computation but not between processes – again, a simple choice, the usefulness of which must be experimentally determined.

5 Application: Coding Events

In π-style communication an output will be received by a single input, whereas in HAN programming it seems that a common idiom will be to broadcast events (some of which are defined by UPnP device descriptions) to many receivers. IOTA does not have primitive support for such events since, as we shall demonstrate here, they can easily be coded up.

One might want event expressions

e	::=	...	
		$!!e.e'$	broadcast event e, with continuation e'
		$??e.e'$	install an event-listener then (after the install has happened) do e'

with typing rules

$$\frac{E \vdash e : \mathsf{MU} \quad E \vdash e' : \mathsf{proc}}{E \vdash !!e.e' : \mathsf{proc}} \qquad \frac{E \vdash e : \mathsf{MU} \to \mathsf{proc} \quad E \vdash e' : \mathsf{proc}}{E \vdash ??e.e' : \mathsf{proc}}$$

that allow XML values to be broadcast ($!!e.e'$) and event-listeners, which are just functions from MU to proc, to be registered. Both have continuations – these are synchronous publish and subscribe. For simplicity we take just a single global 'event channel'.

To encode these in Iota, we start by taking two channels:

$$\begin{array}{lcl} \texttt{publish} & : & (\mathsf{MU} * () \,\mathsf{chan})\,\mathsf{chan} \\ \texttt{listen} & : & ((\mathsf{MU} \to \mathsf{proc}) * () \,\mathsf{chan})\,\mathsf{chan} \end{array}$$

The broadcast and install are encoded as follows:

$$\begin{array}{lcl} [\![!!e.e']\!] & = & \texttt{new}\ z : () \,\mathsf{chan}\ \texttt{in} \\ & & \quad (\texttt{publish!}(e, z)||z?() \Rightarrow e') \\ [\![??e.e']\!] & = & \texttt{new}\ z : () \,\mathsf{chan}\ \texttt{in} \\ & & \quad (\texttt{listen!}(e, z)||z?() \Rightarrow e') \end{array}$$

And an `EventManager` process must be run at top level:

```
EventManager =
  new listeners : (MU → proc) list chan in
    (listeners![]
    ||listen?*(l, z) ⇒ listeners?ls ⇒ (listeners!(l :: ls)||z!())
    ||publish?*(m, z) ⇒ listeners?ls ⇒
      let val F = fn l : MU → proc ⇒ try l m with MatchFailed() ⇒ 0 in
      let val Par = fn x : proc ⇒ fn y : proc ⇒ x||y in
        (foldr Par 0 (map F ls)
        ||listeners!ls
        ||z!())
```

This process maintains state – the list of listener processes – in a private channel `listeners`. When the manager receives a `listen` request, the listener process is simply concatenated to the channel. When the manager receives a `publish` request, it supplies the markup to all the processes waiting on the `listeners` channel. (Note the exception handling code, which implicitly assumes the body of an event-listener function will not raise `MatchFailed` – one might want to be more refined.) This process uses the familiar functions `map` and `foldr`, which are provided in a standard library.

As part of an overall HAN system architecture, conventions on what is evented and what is dealt with by method calls have to be fixed. UPnP events (sic) most device state changes.

6 Application: HAN architecture and controlling a HAN device

Although the details are still evolving, we expect a typical HAN to contain a *home server*. We anticipate that most Iota code will run on the home server, maintaining any required state (e.g. its view of the various home device states) using the standard π idiom of outputs on channels – much in the same way as the `EventManager` process in the previous section. The home server will then communicate to the HAN devices – following the UPnP specifications – using SOAP.

The rest of this section illustrates the code required to control a HAN device – a CD player – that follows the UPnP idiom. The code sends a control message to the CD, querying its volume, then sends another control message to set it to the previous value plus one.

The messages are sent as SOAP invocations, which in turn are embedded in HTTP. SOAP headers are of the form:

```
POST path of control URL   HTTP/1.1
HOST: host of control URL:port of control URL
CONTENT-LENGTH: bytes in body
CONTENT-TYPE: text/xml; charset="utf-8"
SOAPACTION:
"urn:schemas-upnp-org:service:Audio:1#GetAudio"
```

In this section we suppose the path, host and port of the device are contained in a new type DeviceAddress, and wrap up the

```
urn:schemas-upnp-org:service:Audio:1#GetAudio
```

as a value of type SoapAction. We regard the payload of the SOAP request simply as a value of type MU, ignoring any correlation between the SoapAction and the structure of the payload. We suppose a library channel

$$\texttt{invokeSoap} : (\mathsf{DeviceAddress} * \mathsf{SoapAction} * \mathsf{MU} * (\mathsf{MU}\ \mathsf{chan}))\ \mathsf{chan}$$

that sends off SOAP messages in the obvious way (this has not been implemented).

The code can then be written as below. It is regrettably verbose, even with extensive use of our syntactic sugar. However one might reasonably expect there to be UPnP specific libraries, rather than merely SOAP-specific, which would dramatically reduce the size of the code. We revert to non-typeset code and use additional syntactic sugar for `let val` $p = e$ `in` e'.

```
new result : MU chan in     (* make up a new result channel *)
```

```
(invokeSoap!                (* send off the query command, with a 4-tuple of args  *)
   (deviceAddress,
    "urn:schemas-upnp-org:service:Audio:1#GetAudio",
    <s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
                s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"//>
      <s:Body//>
        <u:GetAudio xmlns:u="urn:schemas-upnp-org:service:Audio:1"/>,
    result)              (* this bit of the args is the channel on
                            which we expect the result *)

|result?x=>              (* get result *)
  let val                (* pattern match the result value x *)
    <s:Envelope
         xmlns:s="http://schemas.xmlsoap.org/soap/envelope/",
         s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"//>
       <s:Body//>
         <u:GetAudioResponse xmlns:u="urn:schemas-upnp-org:service:Audio:1"//>
           ( <CurrentVolume>z:int</CurrentVolume> :: *:MU list )
  =x in
    (invokeSoap!         (* send off the set-volume command *)
       (deviceAddress,
        "urn:schemas-upnp-org:service:Audio:1#SetVolume",
        <s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/",
                    s:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"//>
          <s:Body//>
            <u:SetVolume xmlns:u="urn:schemas-upnp-org:service:Audio:1"//>
              <NewVolume> z+1 </NewVolume>,     (* real computation!  *)
        result)
    |result?x=> 0 )                       (* receive ack, then we're done *)
```

7 Application: Networking primitives

The Iota implementation has libraries for sockets programming and for HTTP (the latter written in Iota itself). This makes it very straightforward to, for example, download a piece of XML from the web and extract information from it. Suppose that this XML, with (abridged!) descriptions of 4 HAN devices, is placed on a webserver.

```
<devices>
  <device id="WM01">
    <nature desc="washing machine"/>
    <location value="Kitchen" />
    <manufacturer value = "Bosch"/>
  </device>
  <device id ="HIFI002">
    <nature desc="hifi"/>
    <location value="LivingRoom"/>
    <quiet value="2" />
  <device id ="TSTR007">
   <nature desc="toaster"/>
   <manufacturer value="Siemens Porsche" />
  </device>
  <device id ="TV03">
   <nature desc="television"/>
   <quiet value="3"/>
   <digital provider="NTL"/>
  </device>
```

```
  </device>                              </devices>
```

The Iota code to access it, and return the identifiers of the devices that have a quiet volume (together with the associated values) is below.

```
let fun has_quiet  <quiet value=x /> :: ps => (true,x)
                   |     p : content :: ps => has_quiet ps
                   |                     [] => (false,"")
in let
fun search (<device id=x //>info):: ps => let val (yes,vol) = has_quiet(info)
                                          in if yes
                                             then
                                                (x,vol):: search ps
                                             else search ps
           |                           [] => []
in let
fun findquietparams (<devices>entries</devices>)  => search entries
                    |     *:MU                    => raise UserException()

in new d : content list chan
  in
   Iota.IO.XMLHTTP ("www.cl.cam.ac.uk", 80,
                    "/users/gmb/iota-devices.xml", [], d)
  ||
  d ? fn result => Iota.err ! print(findquietparams result)
```

Similar code has been successfully executed in practice.

This code is extremely simple, which is essentially the point. In the HAN setting, and in other Internet applications, much of the code simply downloads XML and extracts information from it. Pattern matching and simple recursion make this particularly well-suited to functional programming.

8 Conclusion

In this paper we have given an overview of our design of Iota: a concurrent XML scripting language. The novel features of Iota are its approach to integrating XML elements, and its use of channel-based, asynchronous communication primitives to program device behaviour within a home area network. Not least, we have taken a mathematical approach to our language design, providing precise details of the type system and an operational semantics.

It raises several interesting questions. Most obviously, there is the pragmatic question: is the language expressive enough for its intended application, scripting home-area networks? Only experience can tell.

A more general pragmatic question is the extent to which XML found 'in the wild' will conform to DTDs or some form of Schemas, and, if they do, whether within a single domain such as home networking whether particular definitions will become sufficiently widespread to count on. If the answers to these become positive then a richer type system than the one we have presented here, that can express those definitions, will become desirable. On the other hand, if the unstructured status quo continues then there will be a real need for loosely-typed systems such as the one we have presented (or perhaps for optional weakening of the stronger systems). Its simplicity may also be a major advantage, particularly when one thinks of integrating it with the rest of a modern programming language type system.

There is a particular technical question of how to precisely describe the type-based disambiguation that has been implemented. Further, while we have defined the language and operational semantics carefully, we have not attempted to prove type preservation and safety theorems.

An interesting experiment, taking advantage of the small size of the language, would be to reimplement the Iota engine targeting a small virtual machine, such as the KVM.

Acknowledgements

The current implementation[2] of Iota was written, in Java, by Ewan Mellor as part of his undergraduate project. We are grateful to him for his careful coding, and his assistance in the design of Iota. We acknowledge support from a Royal Society University Research Fellowship (Sewell), EPSRC research grant GR/N24872 *Wide-area Programming: Language, Semantics and Infrastructure Design*, and EU grant PEPITO.

[2] Available at `http://www.cl.cam.ac.uk/users/gmb/Iota`

A Iota Definition

A.1 Syntax

We have three classes of syntax: core syntax, derived forms that can be translated out before typechecking (flagged 'derived'), and forms that require type-based disambiguation (flagged 'magic').

We work up to alpha-equivalence throughout, requiring all identifiers in patterns to be distinct and regarding them as binding in the evident scopes.

Types	T	::=	bool	Booleans
			int	Integers
			char	Characters
			string	Strings
			unit	Unit
			$T_1 * .. * T_n$	Tuples $n \geq 2$
			T list	Lists
			$T \rightarrow T$	Function space
			exn	Exceptions
			MU	Mark-up
			content	Content (to be marked up)
			T chan	Channel carrying T
			proc	Processes

Constants and Other Terminals

$\underline{i}$	Integer constant
$\underline{b}$	Boolean constant
$\underline{c}$	Character constant
$\underline{s}$	String constant in quotes, eg "ab"
x	Identifier
ex	Exception constructor
tag	Tag
a	Attribute name

We do not define the concrete syntax tokens for the above here. Characters and strings are Unicode compliant. We assume here that tags and attribute names are taken from a set that is identical to the values of type string. Strictly, this is not so, and we need either a defined injection from strings to tag/attribute names or to raise exceptions dynamically.

We suppose each exception constructor ex has a predetermined type $T(ex)$ of the values it carries. We require there to be a `MatchFailed` constructor, with $T($`MatchFailed`$) =$ unit.

We omit a specification of standard library functions, eg $+ : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and channels for interaction. We would expect there to be library support for timeouts.

Expressions	e ::=	x	Identifier
		$\underline{i}, \underline{b}, \underline{c}, \underline{s}$	Integer, Boolean, Character, String constant
		$()$	Unit
		$(e_1, .., e_n)$	Tuple $n \geq 2$
		$[]$	Empty List
		$e :: e$	Cons
		`if` e `then` e `else` e	Conditional
		`fn` *match*	Function
		`fr` x *match*	Recursive Function
		$e\ e$	Application
		`let` *dec* `in` e	Local declaration
		$ex\,e$	Exception
		`raise` e	Raise exception
		`try` e `with` *match*	Handle exception(s)
		$\langle te\ aes//\rangle e$	Markup
		0	Empty process
		$e \| \| e$	Parallel composition
		`new` $x : T$ `in` e	New channel declaration
		$e!e$	Output along a channel
		$e?e$	Input from a channel
		$e?* e$	Replicated input

and derived forms with their translations:

(e)	$\mapsto$	e	(derived)
$[e_1, .., e_n]$	$\mapsto$	$e_1 :: ..e_n :: []$	(derived) $(n \geq 1)$
$\langle tag\ aes\rangle e\langle/tag\rangle$	$\mapsto$	$\langle tag\ aes//\rangle e$	(derived)
$\langle te\ aes/\rangle$	$\mapsto$	$\langle te\ aes//\rangle[]$	(derived)
$e?match$	$\mapsto$	$e?$`fn` *match*	(derived)
$e?* match$	$\mapsto$	$e?*$ `fn` *match*	(derived)

The ambiguous surface syntax form for markup allows a space-separated sequence of expressions inside a pair of tags:

e	::=	...	
		$\langle tag\ aes\rangle e_1\ ..\ e_n\langle/tag\rangle$	Markup $n \geq 0$ (magic)

These e_i might be of type $\mathsf{content}$, $\mathsf{content\,list}$, $T \to \mathsf{content}$ or indeed any T, depending on their context.

Note there is not a derived form $\langle te\ aes\rangle e\langle/te\rangle$ for an arbitrary te, just for a tag. Moreover,

the tags must match for desugaring to occur. Similarly for patterns below.

Matches	*match*	::=	$p \Rightarrow e$	
			$p \Rightarrow e \mid match$	
Tag expressions	*te*	::=	*tag*	Constant Tag
			$\{e\}$	Computed Tag
Attribute expression sequences	*aes*	::=	empty	Empty sequence
			$a = e\ aes$	

We allow computation of tag names and attribute values, but not of attribute names. This is because (1) it is difficult to statically ensure XML well-formedness with attribute name computation, as there may be repeated attributes; and (2) pragmatically, we expect such computation will not usually be required.

Patterns	p	::=	$* : T$	Wildcard
			$x : T$	Variable
			$\underline{i}, \underline{b}, \underline{c}, \underline{s}$	Integer, Boolean, Character, String constant
			$()$	Unit
			$(p_1, .., p_n)$	Tuple $n \geq 2$
			$[]$	Empty List
			$p :: p$	Cons
			$ex\ p$	Exception constructor pattern
			$\langle tp\ aps//\rangle p$	Markup pattern

and derived forms with their translations:

(p)	$\mapsto$	p	(derived)
$[p_1, .., p_n]$	$\mapsto$	$p_1 :: .. :: p_n :: []$	(derived) $n \geq 1$
$\langle tag\ aps\rangle p\langle/tag\rangle$	$\mapsto$	$\langle tag\ aps//\rangle p$	(derived)
$\langle tp\ aps/\rangle$	$\mapsto$	$\langle tp\ aps//\rangle []$	(derived)

Again we have ambiguous surface syntax for patterns:

p	::=	...	
		$\langle tp\ aps\rangle p_1\ p_2\ ...\ p_n\langle/tp\rangle$	(magic)

Tag Patterns	*tp*	::=	$*$	Wildcard tag pattern
			tag	Constant tag pattern
			$\{x\}$	Identifier
Attribute Patterns	*ap*	::=	$*$	Wildcard
			$\underline{s}$	String
			x	Identifier
Attribute Pattern Sequences	*aps*	::=	empty	Empty sequence
			$*$	Wildcard sequence
			$a = ap\ aps$	

One can envisage richer forms for attributes, eg to pull out a subsequence of attribute definitions, but again that would be hard to statically type. We do not have value equality patterns $= v$ but only the various constant forms. This would be a fairly minor addition if required.

Declarations *dec* ::= `val` $x = e$ Value

with derived form and translation

`fun` x *match* $\mapsto$ `val` $x =$ `fr` x *match* (derived)

We omit syntax for mutually-recursive functions, though that should be provided.

A.2 Typing

Typing and operational semantics are here defined only for the core language.

Type environments E are finite partial functions from identifiers to types. We write E, E' for their union, thereby asserting also that E and E' have disjoint domain.

Judgements

$E \vdash e : T$	under assumptions E, expression e has type T
$E \vdash match : T \to T'$	under assumptions E, match match has type $T \to T'$
$E \vdash te$ ok	under assumptions E, tag expression te is well-formed
$E \vdash aes$ ok	under E, attribute expression sequence aes is well-formed
$\vdash p : T \rhd E'$	pattern p matches type T, giving additional bindings E'
$\vdash tp \rhd E$	tag pattern tp gives bindings E
$\vdash ap \rhd E$	attribute pattern ap gives bindings E
$\vdash aps \rhd E$	attribute pattern sequence aps gives bindings E
$E \vdash dec \rhd E'$	under E, declaration dec gives additional bindings E'
$T <: T'$	type T is a subtype of type T'

$\boxed{E \vdash e : T}$

Data, Functions, Exceptions

$$\frac{E \vdash e : T \quad T <: T'}{E \vdash e : T'}$$

$$\frac{}{E, x : T \vdash x : T} \qquad \frac{}{\begin{array}{l} E \vdash \underline{i} : \mathsf{int} \\ E \vdash \underline{b} : \mathsf{bool} \\ E \vdash \underline{c} : \mathsf{char} \\ E \vdash \underline{s} : \mathsf{string} \\ E \vdash () : \mathsf{unit} \end{array}}$$

$$\frac{E \vdash e_1 : \mathsf{bool} \quad E \vdash e_2 : T \quad E \vdash e_3 : T}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : T} \qquad \frac{E \vdash e_i : T_i \; \underline{i} = 1..n \; n \geq 2}{E \vdash (e_1, .., e_n) : T_1 * .. * T_n}$$

$$\frac{}{E \vdash [] : T \; \mathsf{list}} \qquad \frac{E \vdash e_1 : T \quad E \vdash e_2 : T \; \mathsf{list}}{E \vdash e_1 :: e_2 : T \; \mathsf{list}}$$

$$\frac{E \vdash match : T \to T'}{E \vdash \texttt{fn } match : T \to T'} \qquad \frac{E, x : T \to T' \vdash match : T \to T'}{E \vdash \texttt{fr } x \; match : T \to T'}$$

$$\frac{E \vdash e_1 : T \to T' \quad E \vdash e_2 : T}{E \vdash e_1 e_2 : T'} \qquad \frac{E \vdash dec \rhd E' \quad E, E' \vdash e : T}{E \vdash \texttt{let } dec \texttt{ in } e : T}$$

$$\frac{E \vdash e : T(ex)}{E \vdash ex \; e : \mathsf{exn}}$$

$$\frac{E \vdash e : \mathsf{exn}}{E \vdash \texttt{raise } e : T} \qquad \frac{E \vdash e : T \quad E \vdash match : \mathsf{exn} \to T \quad T \neq \mathsf{proc}}{E \vdash \texttt{try } e \texttt{ with } match : T}$$

XML

$$\frac{E \vdash te \; \mathsf{ok} \quad E \vdash aes \; \mathsf{ok} \quad E \vdash e : \mathsf{content \; list}}{E \vdash \langle te \; aes // \rangle e : \mathsf{MU}}$$

Processes

$$\frac{}{E \vdash 0 : \mathsf{proc}} \qquad \frac{E \vdash e : \mathsf{proc} \quad E \vdash e' : \mathsf{proc}}{E \vdash e || e' : \mathsf{proc}}$$

$$\frac{E, x : T\ \mathsf{chan} \vdash e : \mathsf{proc}}{E \vdash \mathtt{new}\ x : T\ \mathsf{chan}\ \mathtt{in}\ e : \mathsf{proc}}$$

$$\frac{E \vdash e_1 : T\ \mathsf{chan} \quad E \vdash e_2 : T}{E \vdash e_1 ! e_2 : \mathsf{proc}} \qquad \frac{E \vdash e_1 : T\ \mathsf{chan} \quad E \vdash e_2 : T \to \mathsf{proc}}{E \vdash e_1 ? e_2 : \mathsf{proc} \quad E \vdash e_1 ? * e_2 : \mathsf{proc}}$$

Note that typing does not enforce exhaustiveness of matches.

We allow general recursion here, but it may be that primitive recursion would suffice for HAN, expressed with some combinators.

Note we do not allow `try` e_1 `with` *match* for $e_1 : \mathsf{proc}$, as that would require propagating exceptions across threads.

The process part is strictly layered above the functional part – note that `new` $x : T$ `in` e is allowed only for $e : \mathsf{proc}$, and input bodies must be of type $T \to \mathsf{proc}$.

$\boxed{E \vdash match : T \rightarrow T'}$

$$\frac{\vdash p : T \ \triangleright \ E' \quad E, E' \vdash e : T'}{E \vdash p \Rightarrow e : T \rightarrow T'}$$

$$\frac{\vdash p : T \ \triangleright \ E' \quad E, E' \vdash e : T' \quad E \vdash match : T \rightarrow T'}{E \vdash p \Rightarrow e \mid match : T \rightarrow T'}$$

$\boxed{E \vdash te \ \textbf{ok}}$

$$\frac{}{E \vdash tag \ \mathsf{ok}}$$

$$\frac{E \vdash e : \mathsf{string}}{E \vdash \{e\} \ \mathsf{ok}}$$

$\boxed{E \vdash aes \ \textbf{ok}}$

$$\frac{}{E \vdash \mathrm{empty} \ \mathsf{ok}}$$

$$\frac{E \vdash e : \mathsf{string} \quad E \vdash aes \ \mathsf{ok} \quad a \notin aes}{E \vdash a = e \ aes \ \mathsf{ok}}$$

$\boxed{\vdash p : T \ \triangleright \ E'}$

$$\frac{\vdash p : T' \ \triangleright \ E \quad T <: T'}{\vdash p : T \ \triangleright \ E}$$

$$\frac{}{\vdash (* : T) : T \ \triangleright \ \{\}}$$

$$\frac{}{\vdash (x : T) : T \ \triangleright \ x : T}$$

$$\frac{}{\vdash \underline{b} : \mathsf{bool} \ \triangleright \ \{\}}$$

$$\frac{}{\vdash \underline{i} : \mathsf{int} \ \triangleright \ \{\}}$$

$$\frac{}{\vdash \underline{c} : \mathsf{char} \ \triangleright \ \{\}}$$

$$\frac{}{\vdash \underline{s} : \mathsf{string} \ \triangleright \ \{\}}$$

$$\frac{}{\vdash () : \mathsf{unit} \ \triangleright \ \{\}}$$

$$\frac{\vdash p_i : T_i \ \triangleright \ E_i \ \underline{i} = 1..n \ n \geq 2}{\vdash (p_1, .., p_n) : T_1 * .. * T_n \ \triangleright \ E_1, .., E_n}$$

$$\frac{}{\vdash [] : T \ \mathsf{list} \ \triangleright \ \{\}}$$

$$\frac{\vdash p_1 : T \ \triangleright \ E_1 \quad \vdash p_2 : T \ \mathsf{list} \ \triangleright \ E_2}{\vdash p_1 :: p_2 : T \ \mathsf{list} \ \triangleright \ E_1, E_2}$$

$$\frac{\vdash p : T(ex) \ \triangleright \ E'}{\vdash exp : \mathsf{exn} \ \triangleright \ E'}$$

$$\frac{\vdash tp \ \triangleright \ E_1 \quad \vdash aps \ \triangleright \ E_2 \quad \vdash p : \mathsf{content\ list} \ \triangleright \ E_3}{\vdash \langle tp\, aps // \rangle p : \mathsf{MU} \ \triangleright \ E_1, E_2, E_3}$$

$\boxed{\vdash tp \ \triangleright \ E}$

$$\frac{}{\vdash * \ \triangleright \ \{\}} \qquad \frac{}{\vdash tag \ \triangleright \ \{\}} \qquad \frac{}{\vdash \{x\} \ \triangleright \ x : \mathsf{string}}$$

$\boxed{\vdash ap \ \triangleright \ E}$

$$\frac{}{\vdash * \ \triangleright \ \{\}} \qquad \frac{}{\vdash \underline{s} \ \triangleright \ \{\}} \qquad \frac{}{\vdash x \ \triangleright \ x : \mathsf{string}}$$

$\boxed{\vdash aps \ \triangleright \ E}$

$$\frac{}{\vdash \mathrm{empty} \ \triangleright \ \{\}} \qquad \frac{}{\vdash * \ \triangleright \ \{\}} \qquad \frac{\begin{array}{c}\vdash ap \ \triangleright \ E_1 \\ \vdash aps \ \triangleright \ E_2 \\ a \notin aps\end{array}}{\vdash a = ap\, aps \ \triangleright \ E_1, E_2}$$

$\boxed{E \vdash dec \ \triangleright \ E'}$

$$\frac{E \vdash e : T}{E \vdash \mathtt{val}\ x = e \ \triangleright \ x : T}$$

$\boxed{T <: T'}$

$$\frac{}{\begin{array}{l}\mathsf{bool} <: \mathsf{content} \\ \mathsf{int} <: \mathsf{content} \\ \mathsf{char} <: \mathsf{content} \\ \mathsf{string} <: \mathsf{content} \\ \mathsf{MU} <: \mathsf{content}\end{array}} \qquad \frac{}{T <: T} \qquad \frac{\begin{array}{c}T <: T' \\ T' <: T''\end{array}}{T <: T''}$$

$$\frac{T_i <: T'_i \ \underline{i} = 1..n \ n \geq 2}{T_1 * .. * T_n <: T'_1 * .. * T'_n} \qquad \frac{T <: T'}{T \ \mathsf{list} <: T' \ \mathsf{list}} \qquad \frac{\begin{array}{c}T'_1 <: T_1 \\ T_2 <: T'_2\end{array}}{T_1 \to T_2 <: T'_1 \to T'_2}$$

Note the subsumption in the pattern relation.

As usual, $T \, \mathsf{chan}$ is non-variant.

A.3 Operational Semantics

This section defines the reduction semantics only. To specify library channel I/O labelled transitions would be required also.

The operational semantics will only be used for expressions that are typable with respect to a type environment consisting only of channel identifiers. We say a type T is *extensible* if $\exists T'.T = T'\,\mathsf{chan}$, and similarly that a type environment E is extensible if all types in $\mathrm{ran}(E)$ are extensible. We also assume an extensible E_{lib} (with library channels this would grow).

The semantics defines the following sets and relations:

- Values v
- Sequential reduction contexts C
- Concurrent reduction contexts D
- Functional reduction $e_1 \xrightarrow{\texttt{fun}} e_2$
- Structural congruence $e_1 \equiv e_2$
- Process reduction $e_1 \xrightarrow{\texttt{proc}} e_2$
- Combined reduction $e_1 \longrightarrow e_2$

Values

avs	::=	empty	
		$a = \underline{s}\ avs$	if $a \notin \text{attributes}(avs)$
v	::=	x	
		$\underline{i}$	
		$\underline{b}$	
		$\underline{c}$	
		$\underline{s}$	
		$()$	
		$(v_1, .., v_n)$	$n \geq 2$
		$[]$	
		$v :: v$	
		`fn` $match$	
		`fr` x $match$	
		$ex\ v$	
		$\langle tag\ avs//\rangle v$	
		0	
		$v \| \| v$	
		`new` $x : T$ `in` v	
		$v!v$	
		$v?v$	
		$v?* v$	

Note that `raise` v is not a value.

Matching

We define a partial function match($\_, \_, \_$) taking a type environment, a value, and a pattern (in which all variables are distinct) and giving a substitution.

Note that matching involves typing, because of the subtyping with content and MU, and that there may be many types T such that $E \vdash v : T$.

$$
\begin{array}{lcl}
\mathrm{match}(E, v, * : T) & = & \{\} \quad \text{if } E \vdash v : T \\
\mathrm{match}(E, v, x : T) & = & \{v/x\} \quad \text{if } E \vdash v : T \\
\mathrm{match}(E, \underline{i}, \underline{i}) & = & \{\} \\
\mathrm{match}(E, \underline{b}, \underline{b}) & = & \{\} \\
\mathrm{match}(E, \underline{c}, \underline{c}) & = & \{\} \\
\mathrm{match}(E, \underline{s}, \underline{s}) & = & \{\} \\
\mathrm{match}(E, (), ()) & = & \{\} \\
\mathrm{match}(E, (v_1, .., v_n), (p_1, .., p_n)) & = & \mathrm{match}(E, v_1, p_1) \cup ... \cup \mathrm{match}(E, v_n, p_n) \quad n \geq 2 \\
\mathrm{match}(E, [], []) & = & \{\} \\
\mathrm{match}(E, v_1 :: v_2, p_1 :: p_2) & = & \mathrm{match}(E, v_1, p_1) \cup \mathrm{match}(E, v_2, p_2) \\
\mathrm{match}(E, ex\ v, exp) & = & \mathrm{match}(E, v, p) \\
\mathrm{match}(E, \langle tag\ avs // \rangle v, \langle tp\ aps // \rangle p) & = & \mathrm{tmatch}(E, tag, tp) \cup \mathrm{asmatch}(E, avs, aps) \\
 & & \cup \mathrm{match}(E, v, p) \\
\mathrm{match}(E, v, p) & & \text{undefined otherwise}
\end{array}
$$

This definition uses the following auxiliary functions for tag, attribute sequence and attribute matching:

$$
\begin{array}{lcll}
\mathrm{tmatch}(tag, *) & = & \{\} & \\
\mathrm{tmatch}(tag, tag) & = & \{\} & \\
\mathrm{tmatch}\{tag, \{x\}) & = & \{tag/x\} & \\
\mathrm{tmatch}(tag, tag') & & \text{undefined} & \text{if } tag' \neq tag \\
 & & & \\
\mathrm{asmatch}(\mathrm{empty}, \mathrm{empty}) & = & \{\} & \\
\mathrm{asmatch}((a = \underline{s}\ avs), \mathrm{empty}) & & \text{undefined} & \\
\mathrm{asmatch}(avs, *) & = & \{\} & \\
\mathrm{asmatch}(avs, (a = ap\ aps)) & = & \mathrm{amatch}(avs\ a = ap) \cup \mathrm{asmatch}(avs, aps) & \\
 & & & \\
\mathrm{amatch}(\mathrm{empty}, a = ap) & & \text{undefined} & \\
\mathrm{amatch}((a = \underline{s}\ avs), a = ap) & = & \mathrm{amatch}'(\underline{s}, ap) & \\
\mathrm{amatch}((a' = \underline{s}\ avs), a = ap) & = & \mathrm{amatch}(avs, a = ap) & \text{if } a' \neq a \\
 & & & \\
\mathrm{amatch}'(\underline{s}, *) & = & \{\} & \\
\mathrm{amatch}'(\underline{s}, \underline{s}) & = & \{\} & \\
\mathrm{amatch}'(\underline{s}', \underline{s}) & & \text{undefined} & \text{if } \underline{s}' \neq \underline{s} \\
\mathrm{amatch}'(\underline{s}, x) & = & \{\underline{s}/x\} &
\end{array}
$$

Fun-reduction $e_1 \xrightarrow{\texttt{fun}} e_2$

Sequential reduction contexts:

$$
\begin{array}{rcl}
C & ::= & \texttt{if \_ then } e_1 \texttt{ else } e_2 \\
 & & (v_1, .., \_, .., e_n) n \geq 2 \\
 & & \_ :: e \\
 & & v :: \_ \\
 & & \_ e \\
 & & v \_ \\
 & & \texttt{let val } x = \texttt{\_ in } e \\
 & & \texttt{raise \_} \\
 & & \texttt{try \_ with } match \\
 & & \langle \_ aes // \rangle e \\
 & & \langle tag\, a_1 = \underline{s}_1 .. a_m = \_ .. a_n = e_n // \rangle e \\
 & & \langle tag\, avs // \rangle \_ \\
 & & \_ ! e \\
 & & v ! \_ \\
 & & \_ ? e \\
 & & v ? \_ \\
 & & \_ ? * e \\
 & & v ? * \_ \\
 & & \_ || e \\
 & & e || \_ \\
 & & \texttt{new } x : T \texttt{ in \_}
\end{array}
$$

(we use atomic reduction contexts, as the exception propagation rule involves a context equality test).

Axioms:

$$
\begin{array}{lcll}
\texttt{if true then } e_1 \texttt{ else } e_2 & \xrightarrow{\texttt{fun}} & e_1 & \\
\texttt{if false then } e_1 \texttt{ else } e_2 & \xrightarrow{\texttt{fun}} & e_2 & \\
(\texttt{fn } p_1 \Rightarrow e_1 \mid ... \mid p_n \Rightarrow e_n) v & \xrightarrow{\texttt{fun}} & \mathrm{match}(v, p_i) e_i & (1) \\
(\texttt{fr } x\ p_1 \Rightarrow e_1 \mid ... \mid p_n \Rightarrow e_n) v & \xrightarrow{\texttt{fun}} & & (1) \\
 & & \{(\texttt{fr } x p_1 \Rightarrow e_1 \mid ... \mid p_n \Rightarrow e_n)/x\} \mathrm{match}(E_{\mathrm{lib}}, v, p_i) e_i & \\
(\texttt{fn } p_1 \Rightarrow e_1 \mid ... \mid p_n \Rightarrow e_n) v & \xrightarrow{\texttt{fun}} & \texttt{raise MatchFailed}() & (2) \\
(\texttt{fr } x\ p_1 \Rightarrow e_1 \mid ... \mid p_n \Rightarrow e_n) v & \xrightarrow{\texttt{fun}} & \texttt{raise MatchFailed}() & (2) \\
\texttt{let val } x = v \texttt{ in } e & \xrightarrow{\texttt{fun}} & \{v/x\} e & \\
C[\texttt{raise } v] & \xrightarrow{\texttt{fun}} & \texttt{raise } v & (3) \\
\texttt{try raise } v \texttt{ with } p_1 \Rightarrow e_1 \mid .. \mid p_n \Rightarrow e_n & \xrightarrow{\texttt{fun}} & \mathrm{match}(E_{\mathrm{lib}}, v, p_i) e_i & \\
\texttt{try } v \texttt{ with } p_1 \Rightarrow e_1 \mid .. \mid p_n \Rightarrow e_n & \xrightarrow{\texttt{fun}} & v &
\end{array}
$$

(1) where $\underline{i} \in 1..n$ is the least such that $\mathrm{match}(E_{\mathrm{lib}}, v, p_i)$ is defined.

(2) where there is no $\underline{i} \in 1..n$ such that $\mathrm{match}(E_{\mathrm{lib}}, v, p_i)$ is defined.

(3) if there does not exist $(p_1 \Rightarrow e_1 \mid .. \mid p_n \Rightarrow e_n)$ and $\underline{i}$ such that $C = \texttt{try \_ with } p_1 \Rightarrow e_1 \mid .. \mid p_n \Rightarrow e_n$ and $\mathrm{match}(E_{\mathrm{lib}}, v, p_i)$ defined

Note that these rules allow fun-reduction inside expressions of type proc, eg $x!((\texttt{fn } \underline{i} : \mathsf{int} \Rightarrow z!\underline{i})7) \xrightarrow{\texttt{fun}} x!(z!7)$, and even $x!(e \mid (\texttt{fn } \underline{i} : \mathsf{int} \Rightarrow z!\underline{i})7) \xrightarrow{\texttt{fun}} x!(e \mid z!7)$. They do not specify an evaluation order between parallel components (to not over-constrain the implementation). We do specify an evaluation order elsewhere, though. Fun-reduction has no side effects except exceptions, due to the function/process separation enforced by typing (and hence the rules above do not need to deal with scope extrusion).

Structural congruence $e_1 \equiv e_2$

Define a structural equivalence $\equiv$ over core expressions to be the least relation generated by the axioms:

$$\begin{array}{lcll} 0||e_1 & \equiv & e_1 & \\ e_1||e_2 & \equiv & e_2||e_1 & \\ e_1||(e_2||e_3) & \equiv & (e_1||e_2)||e_3 & \\ e_1||\texttt{new } x : T \texttt{ in } e_2 & \equiv & \texttt{new } x : T \texttt{ in } e_1||e_2 & \text{if } x \text{ not free in } e_1 \end{array}$$

with standard rules for equivalence and for congruence with respect to parallel composition and the `new` operator. Note it is important not to have congruence rules for tuples, I/O operators, or any other constructs.

Proc-reduction $e_1 \xrightarrow{\texttt{proc}} e_2$

Concurrent reduction contexts:

$$\begin{array}{lcl} D & ::= & \_ \\ & & \_||e \\ & & \texttt{new } x : T \texttt{ in } D \end{array}$$

Axioms:

$$\begin{array}{lcl} x!v_1||x?v_2 & \xrightarrow{\texttt{proc}} & v_2\,v_1 \\ x!v_1||x?*v_2 & \xrightarrow{\texttt{proc}} & (v_2\,v_1)||x?*v_2 \end{array}$$

Reduction $e \longrightarrow e'$

The complete reduction relation is defined by

$$\frac{e \xrightarrow{\texttt{fun}} e'}{C[e] \longrightarrow C[e']} \qquad \frac{e_1 \equiv D[e_1'] \;\; e_1' \xrightarrow{\texttt{proc}} e_2' \;\; D[e_2'] \equiv e_2}{e_1 \longrightarrow e_2}$$

combining fun reduction and proc reduction, and closing the latter under structural congruence.

Note that the proc rules require the channel and argument parts to both be reduced to values before communication can occur. (In fact, it is uncommon to write e.g. $e!e'$ for non-value e).

Note that typing rules out examples like $x!(\texttt{new}\ y : \mathsf{int}\,\mathsf{chan}\ \texttt{in}\ y)$ where one would have to decide whether to scope-extrude the new before or after the output.

Note that non-handled exceptions in processes here simply become stuck, eg

$$x!(\texttt{raise}\ ex())||x?f$$

The simplest choice here is to report the error on stderr, discard the output or input, and continue executing, for any process structurally congruence to one of the following:

$$\begin{array}{lll} D[\texttt{raise}\ v!e] & D[\texttt{raise}\ v?e] & D[\texttt{raise}\ v?*\ e] \\ D[v!\texttt{raise}\ v'] & D[v?\texttt{raise}\ v'] & D[v?*\ \texttt{raise}\ v'] \end{array}$$

A more satisfactory solution would involve process groups. We do not specify the runtime errors here, but they are straightforward.

References

[ACPP91] M. Abadi, L. Cardelli, B.C. Pierce, and G.D. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on programming languages and systems*, 13(2):237–268, 1991.

[BH01] A.F. Blackwell and R. Hague. AutoHAN: An architecture for programming the home. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 150–157, 2001.

[HP00] H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language (preliminary report). In *International Workshop on the Web and Databases*, volume 1997 of *Lecture Notes in Computer Science*, 2000.

[LC00] D. Lee and W.W. Chu. Comparative analysis of six XML schema languages. *SIGMOD Record*, 29(3):76–87, 2000.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I + II. *Information and Computation*, 100(1):1–77, 1992.

[PT00] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[SGG01] U. Saif, D. Gordon, and D. Greaves. Internet access to a home area network. *IEEE Internet Computing*, 2001.

[SM00] M. Shields and E. Meijer. XMλ: A functional programming language for constructing and manipulating XML documents. Unpublished paper, 2000.

[SWP99] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location-independent communication for mobile agents: a two-level architecture. In *Internet Programming Languages, LNCS 1686*, pages 1–31, October 1999.

[upn00] Understanding Universal Plug and Play (white paper). Available at `http://www.upnp.org`, 2000.

[WR99] M. Wallace and C. Runciman. Haskell and XML: Generic combinations or type-based translation? In *International conference on functional programming*, 1999.