The Design of Distributed Programming Languages

Peter Sewell

University of Cambridge

http://www.cl.cam.ac.uk/users/pes20

With thanks to many co-authors, including: Mair Allen-Williams, Moritz Becker, Gavin Bierman, John Billings, Steve Bishop, Matthew Fairbairn, Pierre Habouzit, Michael Hicks, James Leifer, Iulian Naemtiu, Michael Norrish, Gilles Peskine, Benjamin Pierce, Andrei Serjantov, Mark Shinwell, Michael Smith, Rok Strniša, Asis Unyapoth, Viktor Vafeiadis, Jan Vitek, Keith Wansbrough, Paweł Wojciechowski, Francesco Zappa Nardelli

High-level programming languages

For non-distributed, non-concurrent programming, they're pretty good.

We have ML (SML/OCaml), Haskell, Java, C#, with:

- type safety
- rich concrete types – datatypes and functions
- abstraction mechanisms for program structuring – ML modules with abstract types, type classes and monads, classes and objects,...

High-level programming languages

For non-distributed, non-concurrent programming, they're pretty good.

We have ML (SML/OCaml), Haskell, Java, C#, with:

- type safety
- rich concrete types – datatypes and functions
- abstraction mechanisms for program structuring – ML modules and abstract types, type classes and monads, classes and objects,...

But this is only within *single executions* of *single* programs.

What about *distributed* computation?

Overview

In these talks I aim to introduce some of the main problems in designing languages that are just as good for distributed programming as those are for the local, sequential world. For some we have reasonable solutions; others are still open.

It'll be idiosyncratic, not a survey (but with pointers to other work).

Challenges (1/2)

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (WIPL98,POPL01a)
- Marshalling: choice of distributed abstractions, and trust assumptions (Acute and HashCaml: POPL01b, ICFP03a, ICFP03b, ICFP05, ML06, JFP)
- Dynamic (re)binding and evaluation strategies: exchanging values between programs
- Type equality between programs: run-time type names, type-safe and abstraction-safe interaction (and type equality within programs)
- Typed interaction handles: establishing shared expression-level names between programs
- Version change: type safety in the presence of, and controlling dynamic linking Package and configuration managment?

Challenges (2/2)

- Acute: prototype language, semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml
- Dynamic update: Proteus etc. (ICFP03a again, POPL05; c.f. Hicks)
- Semantics for real-world network abstractions: TCP, UDP, Sockets (TACS01, ESOP02, SIGCOMM05, POPL06)
- Security
  - Security policy: Cassandra (POLICY04, CSFW04)
  - Executing untrusted code: PCC/TAL, secure encapsulation (CSFW99,00), Xen
  - Language-based (Myers, Sabelfeld, Simonet, Zdancewic, etc.);
  - Protocols (Abadi, Fournet, Gordon, Paulson, etc.);
- Module structure again: first-class/recursive/parametric modules. Exposing interfaces to other programs (via communication).

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Local Concurrency

Local: within a single failure domain, within a single trust domain, low-latency interaction.

- Pure (implicit parallelism or skeletons – parallel map, etc.)
- Shared memory
  - mutexes, cvars (incomprehensible, uncomposable, common)
  - transactional (Venari, STM Haskell/Java, AtomCaml,...)
- Message passing

  semantic choices: asynchronous/synchronous, different synchronization styles (CSP/CCS,Join,...), input-guarded/general nondeterministic choice, ...

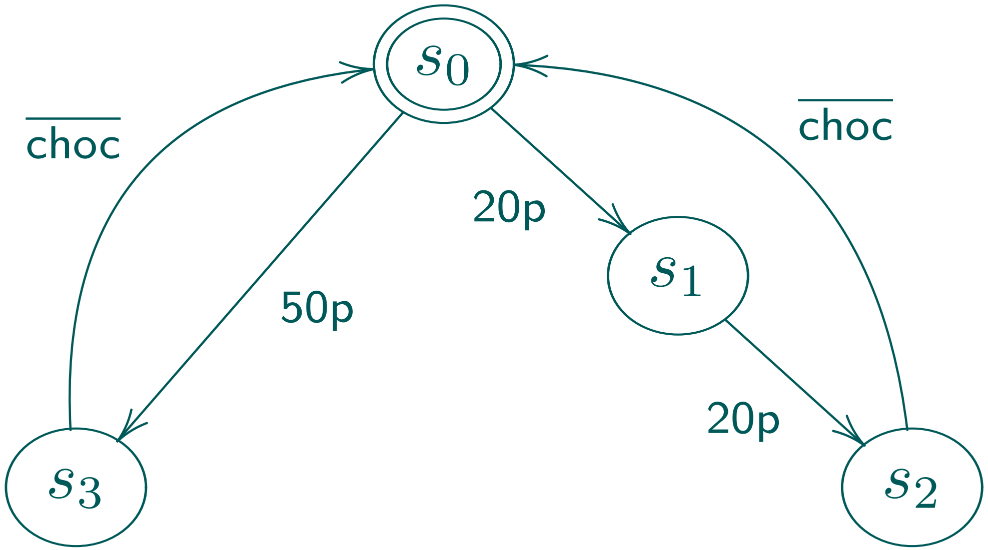
  cf Erlang [AVWW96], Telescript, Facile [TLK96, Kna95], Obliq [Car95], CML* [Rep99], Pict* [PT00], JoCaml [JoC03], Alice [BRS+05] (* concurrent but not distributed)

Process Calculi 1975–1995: Crash Course

1. labelled transition systems
2. equivalences and congruences
3. pure CCS-like calculi, with labelled-transition and reduction semantics
4. value-passing and name-passing
5. π-calculus, with reduction and labelled-transition semantics

Nondeterminacy – Labelled Transition Systems

$\mathcal{S}_{\text{vend}} =$



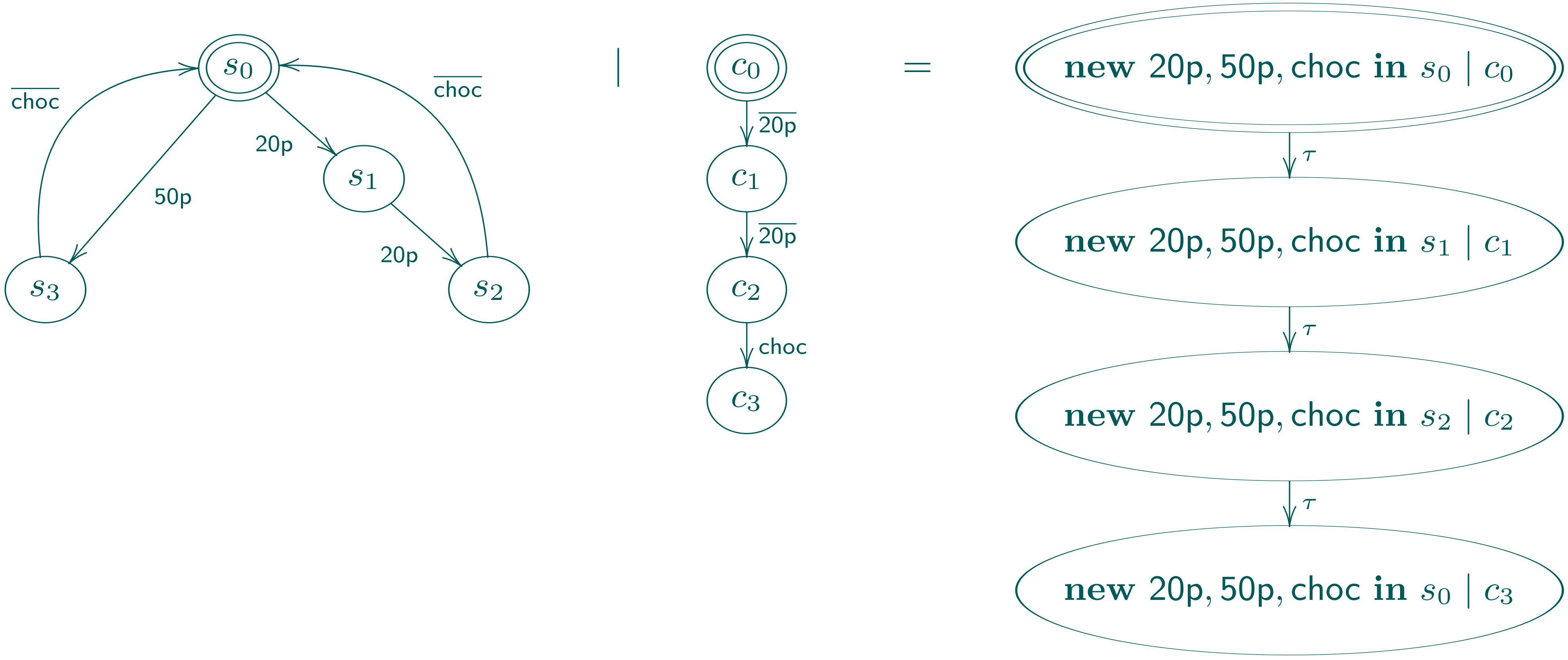
A *labelled transition system* $\mathcal{S}$ is a tuple $\langle L, S, \rightarrow, s_0 \rangle$

- L is a set, of *labels* $L = \{20\text{p}, 50\text{p}, \overline{\text{choc}}\}$
- S is a set, of *states* $S = \{s_0, s_1, s_2, s_3\}$
- $\rightarrow \subseteq S \times L \times S$ is the *transition relation* as above
- $s_0 \in S$ is the *initial state* $s_0 = s_0$

Operations on LTSs – Parallel Composition and Restriction

The vending machine interacting with a sample customer $\mathcal{S}_{\mathsf{cust}}$:

$\mathbf{new}\ \mathsf{20p}, \mathsf{50p}, \mathsf{choc}\ \mathbf{in}\ \mathcal{S}_{\mathsf{vend}} \mid \mathcal{S}_{\mathsf{cust}}$



Parallel Composition and Restriction – Precise Definitions

Fix labels $L = \{\,\overline{n} \mid n \in \mathcal{N}\,\} \cup \{\,n \mid n \in \mathcal{N}\,\} \cup \{\tau\}$ (pure CCS labels).

Define $\mathcal{S} \mid \mathcal{T}$ to have states the pairs $s \mid t$, initial state $s_0 \mid t_0$, and transition relation the least such that:

$$\text{(Par)}\ \frac{s \xrightarrow{\ell} s'}{s \mid t \xrightarrow{\ell} s' \mid t} \qquad\qquad \text{(Com)}\ \frac{s \xrightarrow{n} s' \quad t \xrightarrow{\overline{n}} t'}{s \mid t \xrightarrow{\tau} s' \mid t'}$$

and symmetric rules.

Define $\textbf{new}\ n\ \textbf{in}\ \mathcal{S}$ to have states $\textbf{new}\ n\ \textbf{in}\ s$ for $s \in S$, initial state $\textbf{new}\ n\ \textbf{in}\ s_0$, and transition relation the least such that:

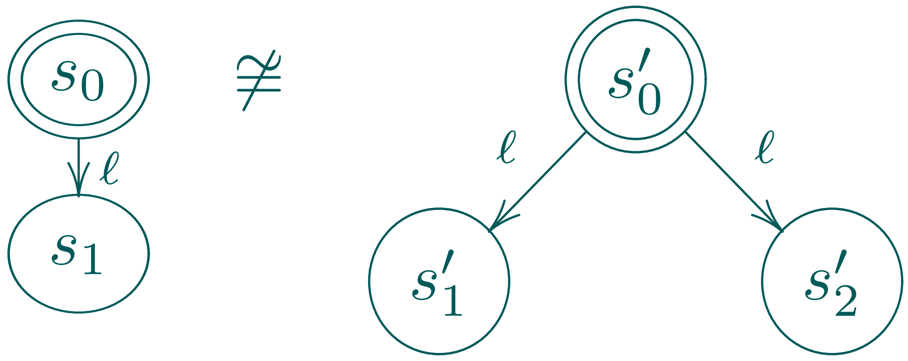
$$\text{(Res)}\ \frac{s \xrightarrow{\ell} s' \quad n \notin \text{fn}(\ell)}{\textbf{new}\ n\ \textbf{in}\ s \xrightarrow{\ell} \textbf{new}\ n\ \textbf{in}\ s'}$$

Equivalences and Congruences

Metatheory: focus on observational congruences and modal logics, not type preservation (until we see fancy π typing)

An LTS is very intensional. Quotienting to factor out excess structure:

LTS isomorphism deals with node identity, but



Partial traces say $\mathcal{S} \simeq_{\mathsf{ptr}} \mathcal{S}' \iff \mathrm{ptr}(\mathcal{S}) = \mathrm{ptr}(\mathcal{S}')$, where

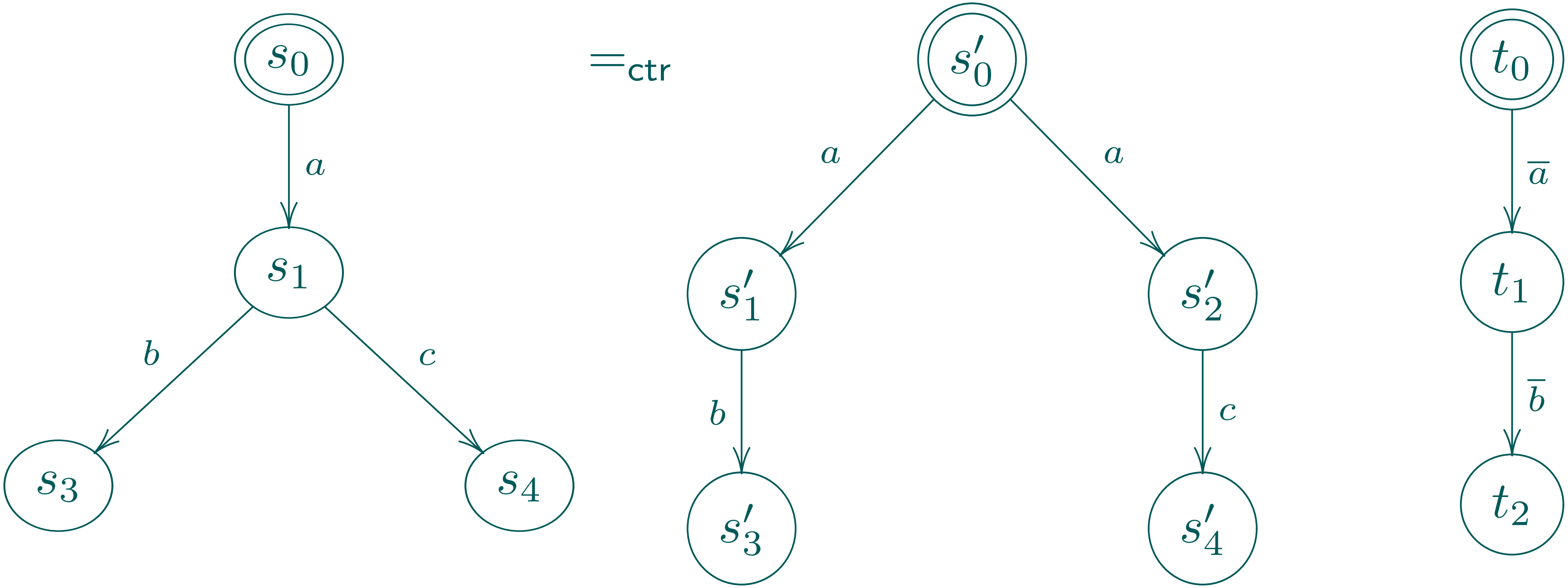
$\mathrm{ptr}(\mathcal{S}) = \{\, \ell_1 \ldots \ell_n \mid \exists s_1, \ldots, s_n \,.\, s_0 \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_n} s_n \,\}$

but this ignores termination/deadlock...

Completed traces

$$\begin{array}{rcl} \mathrm{ctr}(\mathcal{S}) & = & \{\, \ell_1 \ldots \ell_n \mid \exists s_1, \ldots, s_n \,.\, s_0 \xrightarrow{\ell_1} s_1 \cdots \xrightarrow{\ell_n} s_n \not\longrightarrow \} \\ & \cup & \{\, \ell_1 \ell_2 \ldots \mid \exists s_1, s_2, \ldots \,.\, s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \cdots \} \end{array}$$

but not a congruence for CCS parallel:



Strong Bisimulation Say $\sim \subseteq S \times T$ is *a* strong bisimulation if for all (s, t) such that $s \sim t$ we have

- if $s \xrightarrow{\ell} s'$ then $\exists t' \,.\, t \xrightarrow{\ell} t' \wedge s' \sim t'$
- if $t \xrightarrow{\ell} t'$ then $\exists s' \,.\, s \xrightarrow{\ell} s' \wedge s' \sim t'$

henceforth write $\sim$ for the largest such.

Theorem 1.1 Congruence of $\sim$ for $|$

$$(\text{Par})\ \frac{s \sim s' \quad t \sim t'}{s \mid t \sim s' \mid t'}$$

$$\begin{array}{c} \cong \\ | \\ \sim \\ | \\ =_{\mathsf{ctr}} \\ | \\ =_{\mathsf{ptr}} \end{array}$$

Diversion: Alternatives

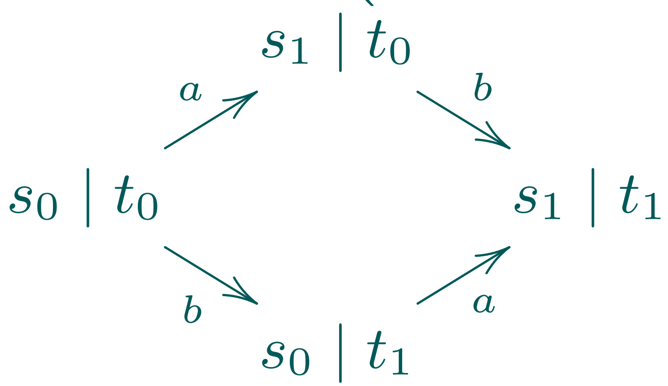
Other synchronisation primitives Take labels $L = \mathcal{N}$ and $A \subseteq \mathcal{N}$

$$(\text{Com})\ \frac{s \xrightarrow{a} s' \quad t \xrightarrow{a} t' \quad a \in A}{s \,||_A\, t \xrightarrow{a} s' \,||_A\, t'}$$

CSP style (consider $\mathcal{S} \,||_A\, \mathcal{T} \,||_A\, \mathcal{U}$). cf modelling vs programming.

Noninterleaving If $\mathcal{S} = \left(s_0 \xrightarrow{\quad a \quad} s_1 \right)$ and $\mathcal{T} = \left(t_0 \xrightarrow{\quad b \quad} t_1 \right)$ then

$\mathcal{S} \mid \mathcal{T} =$



emphasising that LTSs and the definition of $\mid$ reduce parallelism to nondeterministic choices between all interleavings. Sometimes convenient to build more data into the model, cf Event Structures.

Calculi and SOS (Structural Operational Semantics)

So far, been *model-theoretic* – LTSs could have arbitrary sets and relations.

Instead, can consider processes $\mathcal{P}$ defined by a syntax, e.g..

P, Q	$::=$	0	nil
		$P \mid Q$	parallel composition of P and Q
		$\ell.P$	do ℓ then be P
		$\textbf{new } n \textbf{ in } P$	restrict n
		$P + Q$	nondeterministic choice between the actions of P and Q

where the prefixes ℓ are n, $\overline{n}$, or τ (exactly the labels from before).

Now define transition relations inductively:

$$\text{(Par)}\ \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \text{(Com)}\ \frac{P \xrightarrow{n} P' \quad Q \xrightarrow{\overline{n}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{(Res)}\ \frac{P \xrightarrow{\ell} P' \quad n \notin \mathrm{fn}(\ell)}{\mathbf{new}\ n\ \mathbf{in}\ P \xrightarrow{\ell} \mathbf{new}\ n\ \mathbf{in}\ P'}$$

$$\text{(Pre)}\ \frac{}{\ell.P \xrightarrow{\ell} P} \qquad \text{(Sum)}\ \frac{P \xrightarrow{\ell} P'}{P + Q \xrightarrow{\ell} P'}$$

and symmetric rules. Now $\langle L, \mathcal{P}, \rightarrow, P \rangle$ is an LTS for any process term $P \in \mathcal{P}$.

Reduction Semantics

A different way to define the τ transition relation. For example, for $P, Q ::= 0|P \mid Q|\ell.P$, instead of

$$\text{(Pre)}\ \frac{}{\ell.P \xrightarrow{\ell} P} \qquad \text{(Par)}\ \frac{P \xrightarrow{\ell} P'}{P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \text{(Com)}\ \frac{P \xrightarrow{n} P' \quad Q \xrightarrow{\overline{n}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

we define *structural congruence* as the least congruence s.t.

$$P \mid 0 \equiv P \qquad P \mid Q \equiv Q \mid P \qquad P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

and *reduction* by

$$\text{(Par)}\frac{P \to P'}{P \mid Q \to P' \mid Q} \quad \text{(Com)}\frac{}{n.P \mid \overline{n}.Q \to P \mid Q} \quad \text{(Str)}\frac{P \equiv P' \to P'' \equiv P'''}{P \to P'''}$$

Theorem 1.2 $P \to Q$ iff $P \xrightarrow{\tau} \equiv Q$.

Value Passing

Allow channels to carry values, so instead of pure outputs $\overline{n}.P$ and inputs $n.Q$ allow e.g.. $\overline{n}\langle 15, 3\rangle.P$ and $n\langle x_1, x_2\rangle.Q$.

Value 6 being sent along channel x:

$$\overline{x}6 \mid xu.\overline{y}u \quad \xrightarrow{\tau} \quad \{6/u\}(\overline{y}u) = \overline{y}6$$

Tuple values (and tuple patterns):

$$\overline{x}\langle 8, 3\rangle \mid x\langle z_1, z_2\rangle.\overline{y}z_1 \quad \xrightarrow{\tau} \quad \{\langle 8,3\rangle/\langle z_1, z_2\rangle\}(\overline{y}z_1) = \overline{y}8$$

Many outputs on the same channel competing for the same input:

$$\overline{x}5 \mid \overline{x}6 \mid xu.\overline{y}u \quad \nearrow \quad \overline{x}6 \mid \overline{y}5$$

$$\overline{x}5 \mid \overline{x}6 \mid xu.\overline{y}u \quad \searrow \quad \overline{x}5 \mid \overline{y}6$$

Many inputs on the same channel competing for an output:

$$\overline{x}5 \mid xu.\overline{y}u \mid xu.\overline{z}u \quad \begin{array}{l} \nearrow \ \overline{y}5 \mid xu.\overline{z}u \\ \searrow \ xu.\overline{y}u \mid \overline{z}5 \end{array}$$

(do we really want all this expressiveness for programming?)

Restricted names are different from all others:

$$\begin{array}{ccc} \overline{x}5 \mid \mathbf{new}\ x\ \mathbf{in}\ (\overline{x}6 \mid xu.\overline{y}u) & \longrightarrow & \overline{x}5 \mid \mathbf{new}\ x\ \mathbf{in}\ \overline{y}6 \\ \| & & \| \\ \overline{x}5 \mid \mathbf{new}\ x'\ \mathbf{in}\ (\overline{x'}6 \mid x'u.\overline{y}u) & \longrightarrow & \overline{x}5 \mid \mathbf{new}\ x''\ \mathbf{in}\ \overline{y}6 \end{array}$$

(note that we're working with alpha equivalence classes)

Name passing

Now allow those values to include channel names.

The π calculus: Milner, Parrow, Walker 92

A name received on a channel can then be used itself as a channel name for output or input – here y is received on x and then used to output 7:

$$\overline{x}y \mid xu.\overline{u}7 \quad \longrightarrow \quad \overline{y}7$$

Finally, a restricted name can be sent outside its original scope. Here y is sent on channel x outside the scope of the $\mathbf{new}\ y\ \mathbf{in}$ binder, which must therefore be moved (with care, to avoid capture of free instances of y). This is *scope mobility*:

$$\begin{aligned}(\mathbf{new}\ y\ \mathbf{in}\ \overline{x}y \mid yv.P) \mid xu.\overline{u}7 \quad &\longrightarrow \quad \mathbf{new}\ y\ \mathbf{in}\ yv.P \mid \overline{y}7 \\ &\longrightarrow \quad \mathbf{new}\ y\ \mathbf{in}\ \{7/v\}P\end{aligned}$$

$$
\begin{aligned}
(\mathbf{new}\ y\ \mathbf{in}\ \overline{x}y \mid yv.P) \mid xu.\overline{u}7 &\equiv (\mathbf{new}\ y\ \mathbf{in}\ \overline{x}y \mid yv.P \mid xu.\overline{u}7) \\
&\longrightarrow \mathbf{new}\ y\ \mathbf{in}\ yv.P \mid \overline{y}7 \\
&\longrightarrow \mathbf{new}\ y\ \mathbf{in}\ \{7/v\}P
\end{aligned}
$$

The Simplest π-Calculus: Reduction Semantics

Syntax

P, Q	$::=$	0	nil
		$P \mid Q$	parallel composition of P and Q
		$\overline{c}v$	output v on channel c
		$cw.P$	input from channel c
		$\textbf{new } c \textbf{ in } P$	new channel name creation

Structural Congruence

$$
\begin{aligned}
P \mid 0 &\equiv P \\
P \mid Q &\equiv Q \mid P \\
P \mid (Q \mid R) &\equiv (P \mid Q) \mid R \\
\textbf{new } x \textbf{ in new } y \textbf{ in } P &\equiv \textbf{new } y \textbf{ in new } x \textbf{ in } P \\
P \mid \textbf{new } x \textbf{ in } Q &\equiv \textbf{new } x \textbf{ in } (P \mid Q) \qquad x \notin \mathrm{fn}(P)
\end{aligned}
$$

Reduction

(Com) $$\overline{\overline{c}v \mid cw.P \rightarrow \{v/w\}P}$$

(Par) $$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q}$$

(Res) $$\frac{P \rightarrow P'}{\textbf{new } x \textbf{ in } P \rightarrow \textbf{new } x \textbf{ in } P'}$$

(Struct) $$\frac{P \equiv P' \rightarrow P'' \equiv P'''}{P \rightarrow P'''}$$

Focus on name creation, using scope extrusion as semantic tool for locally generating globally fresh names (could do gensym semantics, but more awkward).

Expressiveness

A small calculus (and the semantics only involves name-for-name substitution, not term-for-variable substitution), but very expressive:

- encoding data structures
- encoding functions as processes (Milner, Sangiorgi)
- encoding higher-order π in π (Sangiorgi)
- encoding synchronous communication with asynchronous (Honda/Tokoro, Boudol)
- encoding polyadic communication with monadic (Quaglia, Walker)
- encoding choice (or not) (Nestmann, Palamidessi)
- ...

Modelling vs Programming: Choices of Primitives

- Monadic
- No replicated input $*xy.P$ or full replication $!P \equiv P \,|!P$
- No name (in)equality testing (natural to have in a programming lang)
- No $+$ (don't seem to need arbitrary $P + Q$ for programming; can code some choice)
- Asynchronous (fits with asychronous messaging. Can encode synchronous communication – in some sense)
- No process-passing, or higher order values

Facile, CML, Pict, ...

Programming: Pict (Pierce, Turner) [PT00]

An experimental concurrent (not distributed) programming language based on the π calculus.

Process abstractions:

$$
\begin{array}{l}
\textbf{new}\ \mathit{plustwo}\ \textbf{in} \\
\quad *\mathit{plustwo}\langle x\ r\rangle.\overline{r}(x+2) \\
\quad |\ \textbf{new}\ r\ \textbf{in} \\
\qquad \overline{\mathit{plustwo}}\langle 56\ r\rangle\ |\ rz.\overline{\mathit{printi}}\,z
\end{array}
$$

Locks and methods:

$$\begin{array}{l}
\textbf{new}\ \mathit{lock}\ \textbf{in} \\
\quad \overline{\mathit{lock}}\langle\rangle \\
\quad \mid *\mathit{method1}\ \mathit{arg}. \\
\qquad \mathit{lock}\langle\rangle. \\
\qquad\quad \ldots \\
\qquad\quad \overline{\mathit{lock}}\langle\rangle \\
\\
\quad \mid *\mathit{method2}\ \mathit{arg}. \\
\qquad \mathit{lock}\langle\rangle. \\
\qquad\quad \ldots \\
\qquad\quad \overline{\mathit{lock}}\langle\rangle
\end{array}$$

Objects: $\langle \mathit{method1}\ \ \mathit{method2} \rangle$

The Simplest π-Calculus: Labelled Transition Semantics

The labelled transition relation has the form $A \vdash P \xrightarrow{\ell} Q$ where A is a finite set of names, $\mathrm{fn}(P) \subseteq A$, and ℓ is from

$$
\begin{array}{lll}
\ell & ::= & \tau & \text{internal action} \\
 & & \overline{x}v & \text{output of } v \text{ on } x \\
 & & xv. & \text{input of } v \text{ on } x
\end{array}
$$

Output of a free name:

$$\{x, y\} \vdash \overline{x}y \xrightarrow{\overline{x}y} 0$$

Output of a new name (for *any* $w \neq x$):

$$\{x\} \vdash \mathbf{new}\ \hat{y}\ \mathbf{in}\ \overline{x}\hat{y} \xrightarrow{\overline{x}w} 0$$

Input of a name:

$$A \vdash xu.P \xrightarrow{xw.} \{w/u\}P$$

SOS rules

$$\text{(Out)}\ \frac{}{A \vdash \overline{x}v \xrightarrow{\overline{x}v} 0} \qquad \text{(In)}\ \frac{}{A \vdash xp.P \xrightarrow{xv.} \{^v/_p\}P}$$

$$\text{(Par)}\ \frac{A \vdash P \xrightarrow{\ell} P'}{A \vdash P \mid Q \xrightarrow{\ell} P' \mid Q} \qquad \text{(Com)}\ \frac{A \vdash P \xrightarrow{\overline{x}v} P' \quad A \vdash Q \xrightarrow{xv.} Q'}{A \vdash P \mid Q \xrightarrow{\tau} \textbf{new}\ \{v\} - A\ \textbf{in}\ (P' \mid Q')}$$

$$\text{(Open)}\ \frac{A, x \vdash P \xrightarrow{\overline{y}x} P' \quad y \neq x}{A \vdash \textbf{new}\ x\ \textbf{in}\ P \xrightarrow{\overline{y}x} P'} \qquad \text{(Res)}\ \frac{A, x \vdash P \xrightarrow{\ell} P' \quad x \notin \text{fn}(\ell)}{A \vdash \textbf{new}\ x\ \textbf{in}\ P \xrightarrow{\ell} \textbf{new}\ x\ \textbf{in}\ P'}$$

$$\text{(Struct Right)}\ \frac{A \vdash P \xrightarrow{\ell} P' \quad P' \equiv P''}{A \vdash P \xrightarrow{\ell} P''}$$

Structural congruence on the left of a transition is admissible, and the reduction and transition semantics give exactly the same internal steps.

Theorem 1.3 If $P' \equiv P$ then $A \vdash P' \xrightarrow{\ell} Q$ iff $A \vdash P \xrightarrow{\ell} Q$.

Theorem 1.4 If $\text{fn}(P) \subseteq A$ then $P \to Q$ iff $A \vdash P \xrightarrow{\tau} Q$.

Scope extrusion (example from Slide 22):

$$\text{(Out)}\ \dfrac{}{\{x,y\} \vdash \overline{x}y \xrightarrow{\overline{x}y} 0}$$

$$\text{(Par)}\ \dfrac{\{x,y\} \vdash \overline{x}y \xrightarrow{\overline{x}y} 0}{\{x,y\} \vdash \overline{x}y \mid yv.P \xrightarrow{\overline{x}y} 0 \mid yv.P}$$

$$\text{(Open)}\ \dfrac{\{x,y\} \vdash \overline{x}y \mid yv.P \xrightarrow{\overline{x}y} 0 \mid yv.P}{\{x\} \vdash \mathbf{new}\ \hat{y}\ \mathbf{in}\ \overline{x}\hat{y} \mid \hat{y}v.P \xrightarrow{\overline{x}y} 0 \mid yv.P} \quad \text{(In)}\ \dfrac{}{\{x\} \vdash xy.\overline{u}7 \xrightarrow{xy.} \{y/u\}\overline{u}}$$

$$\text{n)}\ \dfrac{\cdots}{\{x\} \vdash (\mathbf{new}\ \hat{y}\ \mathbf{in}\ \overline{x}\hat{y} \mid \hat{y}v.P) \mid (xy.\overline{u}7) \xrightarrow{\tau} \mathbf{new}\ \hat{\hat{y}}\ \mathbf{in}\ 0 \mid \hat{\hat{y}}v.P \mid \overline{\hat{\hat{y}}}7}$$

π Equivalences and Congruences

Partial traces

$$\mathrm{ptr}_A(P) = \{\, \ell_1 \ldots \ell_n \mid A \vdash P \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n} \}$$

Strong Bisimulation Take *bisimulation* $\dot{\sim}$ to be the largest family of relations indexed by finite sets of names such that each $\dot{\sim}_A$ is a relation over $\{\, P \mid \mathrm{fn}(P) \subseteq A \,\}$ and for all $P \mathrel{\dot{\sim}_A} Q$,

- if $A \vdash P \xrightarrow{\ell} P'$ then $\exists Q' \,.\, A \vdash Q \xrightarrow{\ell} Q' \wedge P' \mathrel{\dot{\sim}_{A \cup \mathrm{fn}(\ell)}} Q'$
- if $A \vdash Q \xrightarrow{\ell} Q'$ then $\exists P' \,.\, A \vdash P \xrightarrow{\ell} P' \wedge P' \mathrel{\dot{\sim}_{A \cup \mathrm{fn}(\ell)}} Q'$

Theorem 1.5 Bisimulation $\dot{\sim}$ is an indexed congruence.

Define $\sim$ by $P \sim_A Q$ iff for all substitutions σ with $\mathrm{dom}(\sigma) \cup \mathrm{ran}(\sigma) \subseteq A$ we have $\sigma P \mathrel{\dot{\sim}_A} \sigma Q$.

Typing

Simple typing:

$$T ::= \ldots | T \text{ chan}$$

IO subtyping, Linear typing, Polymorphism, ...
(Honda, Kobayashi, Odersky, Pierce, Sangiorgi, Yoshida,...)

Adapting typing from functional languages – reasonably straightforward.

Knock-on effects on observational congruences – interesting!

Typing for subtle behavioural properties, e.g. deadlock freedom – interesting!

Pointers

There are many subtle technical choices in how one sets up a π-calculus semantics. This presentation is based on

> Applied Pi — A Brief Tutorial. Peter Sewell. Technical Report 498, Computer Laboratory, University of Cambridge 2000 `http://www.cl.cam.ac.uk/users/pes20/apppi.ps`.

The mobility web page `http://lamp.epfl.ch/mobility/` has pointers to several other introductory tutorials, and to the books by Milner and by Sangiorgi and Walker. See also forthcoming CONCUR 06 tutorial by Nestmann.

Reflections: Asynchronous π – a good fit to distributed systems?

+ clear treatment of concurrency
+ asynchronous π communication not far above real comms
+ π-style naming is widely applicable
  - communication channels (with read/write operations)
  - cryptographic keys (with decrypt/encrypt)
  - reference cells (with deref/assign)
  - process groups
  - nonces
  - type ids

  (all are dynamically and locally generable pure names) but it doesn't address:

- point-to-point or multicast comms
- failure (machines and comms); timeouts, transactions
- security properties (secrecy, integrity, authenticity, non-repudiation, anonymity) and their implementation using crypto
- secure encapsulation; policy managment
- code, computation and device mobility
- performance

and

- proofs of concurrent algorithms are still difficult
- asynchronous π communication is far above real comms

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

The Distributed Process Calculi of the late 90s

- π_l calculus (Amadio, Prasad, 94) [AP94], modelling the failure semantics of Facile [TLK96, Kna95] (Thomson et al).
- Distributed Join Calculus (Fournet et al, 96) [FGL$^+$96], as the basis for a mobile computation language.
- Spi calculus (Abadi, Gordon 97) [AG97], for reasoning about security protocols.
- dpi calculus (Sewell, 98) [Sew98], with locality enforcement of capabilities with a subtyping system.
- Nomadic π calculus (Sewell, Unyapoth, Wojciechowski, Pierce 98) [SWP98b], studying communication infrastructures for mobile computations.
- Ambient calculus (Cardelli, Gordon 98) [CG98], modelling security domains.
- Seal calculus (Vitek, Castagna 98) [VC98], focussing on protection mechanisms including revocable capabilities.
- Box-π (Sewell, Vitek 99) [SV99, SV00], secure encapsulation of untrusted components and causality typing.
- Dπ calculus (Riely, Hennessy, 99) [RH99], typing for open systems of mobile computations.

Grouping

π has *names* and *processes* (but they don't have identity).

May want to add primitives for *grouping* process terms, into units of:

- failure (e.g.. machines or runtime system instances);
- migration (e.g.. mobile computations);
- trust (e.g.. large administrative domains or small secure critical regions);
- synchronisation (i.e.. regions within which an output and an input on the same channel name can interact).

Mobility

- Scope mobility (as in π). Fundamental.
- Code mobility. Fundamental. Varying timescales: deployment/runtime.
- Computation mobility. Late 90s fashion?

  In-the-small: value unclear. In-the-large: key management win?
  But might be as OS level (cf Xen, VMWare) rather than language level?

  Nonetheless, focus here for the rest of this lecture — both for itself and as a source of motivating examples of distributed abstractions.
- Device mobility. More networking issues than PL issues? (but note the implicit crossings of trust boundaries)

Computation Mobility – DJoin and JoCaml [FGL+96, JoC03]

The Distributed Join Calculus (Fournet, Gonthier, Lévy, Maranget, Rémy).

Take a tree of *locations*, each uniquely named. Generable.

Migration allows any location ℓ to move to become a child of any non-descendant.

Join patterns combine restriction, replicated input and linearity – $\mathbf{def}\ x(w).P\ \mathbf{in}\ Q$ is similar to $\mathbf{new}\ x\ \mathbf{in}\ Q \mid *xw.P$. An example reduction:

$$\begin{array}{rl} & \mathbf{def}\ (x_1(w_1) \wedge x_2(w_2)).P\ \mathbf{in}\ Q \mid \overline{x_1}3 \mid \overline{x_2}7 \\ \rightarrow & \mathbf{def}\ (x_1(w_1) \wedge x_2(w_2)).P\ \mathbf{in}\ Q \mid \{3,7/w_1,w_2\}P \end{array}$$

Synchronization conjunctions and disjunctions for expressiveness (encodings to and from pi).

JoCaml: programming language implementation based on Join

Reflections

Communication is location-independent (unique loc to deliver to) – complex distributed infrastructure built in to implementation, for forwarding and also for distributed GC (Le Fessant).

Hence the possible failure models are either simple, with forced kills (but not perhaps useful), or very complex, reflecting what happens to that infrastructure when part fails (not reflected in the semantics).

Computation Mobility – Nomadic π and Nomadic Pict

(Sewell, Unyapoth, Wojciechowski, Pierce. 1997–2001)

Focus on distributed communication infrastructure for mobility. Two kinds of communication:

- Low-level *location-dependent* (LD) primitives, that require an programmer to know the current site of a mobile computation in order to communicate with it. Easy to implement.
- High-level *location-independent* (LI) primitives, that allow communication with a mobile computation irrespective of its current site. This needs subtle distributed infrastructure algorithms.

Questions:

- how can we express these distributed algorithms clearly?
- different algorithms have very different performance and robustness properties – what's the algorithm design space?
- how do we reason about them, to prove them correct?

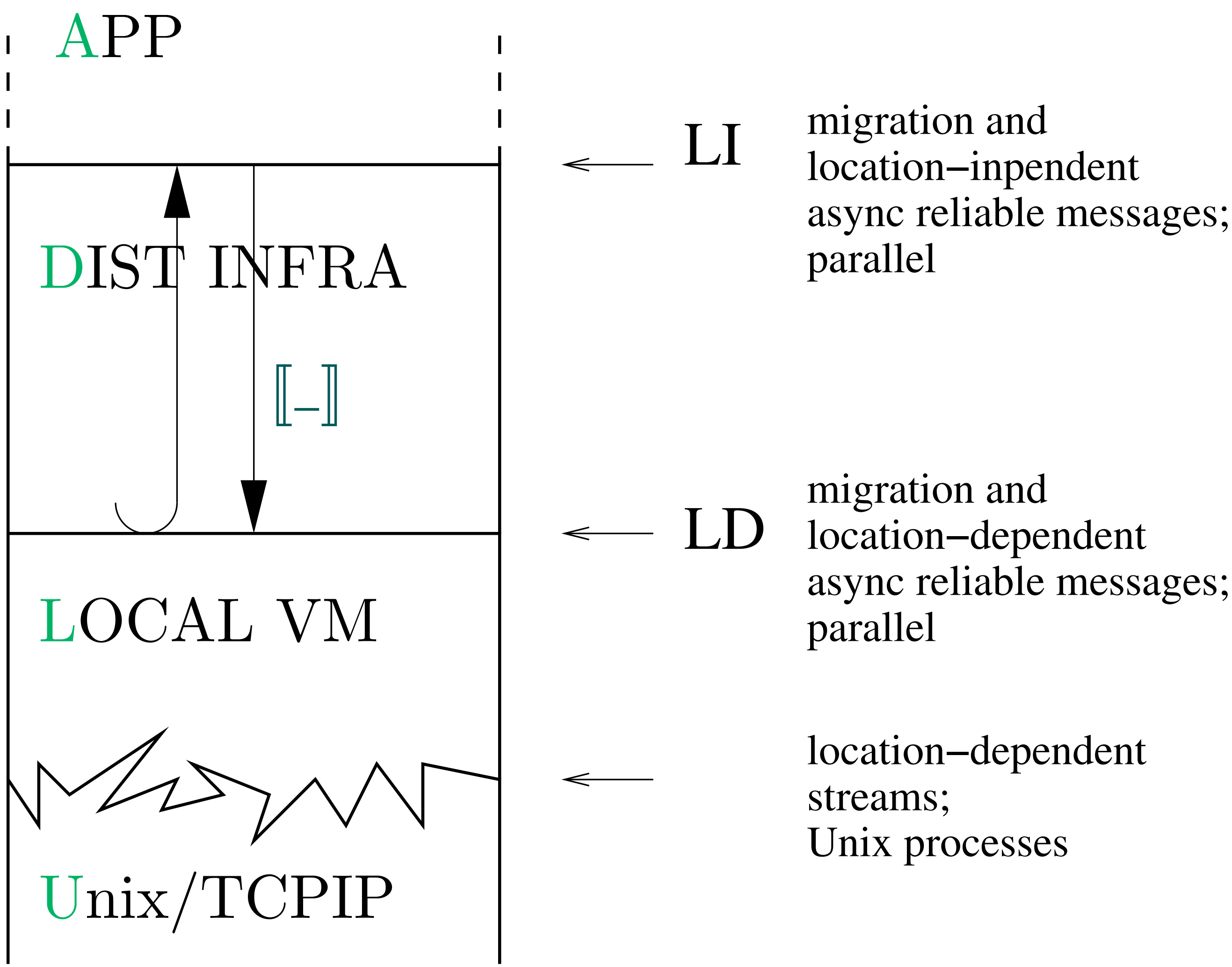
Approach: design smallest calculus that's rich enough to express the algorithms and the application programs that use them.

Nomadic π allows such algorithms to be expressed as translations $[\![\_]\!]$ of the whole calculus, including location-independent communication, into the fragment with only location-dependent communication.

We implemented a corresponding Nomadic Pict distributed programming language, prototyped many algorithms, and proved one of them correct.

Take a short tree of *sites* and *agents*, each uniquely named. Agents are located at sites, and are dynamically generable. Each contains a pi-like process.

Migration allows any agent a to move to become a child of any site s.



The Nomadic π-Calculus

Take names of sites s, of agents a, b, and of channels c. Processes P, Q are as below.

Low-Level:

Agents

$\mathbf{agent}\ a = P\ \mathbf{in}\ Q$	agent creation
$\mathbf{migrate\ to}\ s.P$	agent migration
$P \mid Q$	parallel composition
0	nil

π-calculus-style communication within agents

$\mathbf{new}\ c\ \mathbf{in}\ P$	new channel name creation
$\overline{c}v$	output v on channel c in the current agent
$cw.P$	input from channel c
$*cw.P$	replicated input from channel c
$\mathbf{if}\ a = b\ \mathbf{then}\ P\ \mathbf{else}\ Q$	name equality testing

Inter-agent communication

$\langle a@s \rangle c!v$	LD output to agent a on site s
$\mathbf{iflocal}\ \overline{\langle a \rangle c}v.P\ \mathbf{else}\ Q$	test-and-send to agent a on current site

High-Level: . . .all the above and:

$\overline{\langle a@? \rangle c}v$	LI output to agent a

Reduction Semantics

π-style communication, but synchronisation only within the same agent.

Write $@_a\, P$ for P as part of agent a. Record the current sites of agents in σ.

Low level is implementable with ≤ 1 async inter-site message / reduction.

Low-Level:

$$\sigma; @_a\, \mathbf{agent}\ b = P\ \mathbf{in}\ Q \quad \longrightarrow \quad \sigma; \mathbf{new}\ b@\sigma(a)\ \mathbf{in}\ (@_b\, P | @_a\, Q)$$

$$\sigma; @_a\, \mathbf{migrate\ to}\ s.P \quad \longrightarrow \quad (\sigma \oplus a \mapsto s), @_a\, P$$

$$\sigma; @_a\, \langle b@s \rangle c!v \quad \longrightarrow \quad \sigma; @_b\, \overline{c}v \qquad \text{if } \sigma(b) = s$$

$$\sigma; @_a\, \langle b@s \rangle c!v \quad \longrightarrow \quad \sigma; 0 \qquad \text{if } \sigma(b) \neq s$$

$$\sigma; @_a\, (\overline{c}v | cp.P) \quad \longrightarrow \quad \sigma; @_a\, \{v/p\}P$$

$$\sigma; @_a\, \mathbf{iflocal}\ \overline{\langle b \rangle c}v.P\ \mathbf{else}\ Q \quad \longrightarrow \quad \sigma; @_b\, \overline{c}v | @_a\, P \qquad \text{if } \sigma(a) = \sigma(b)$$

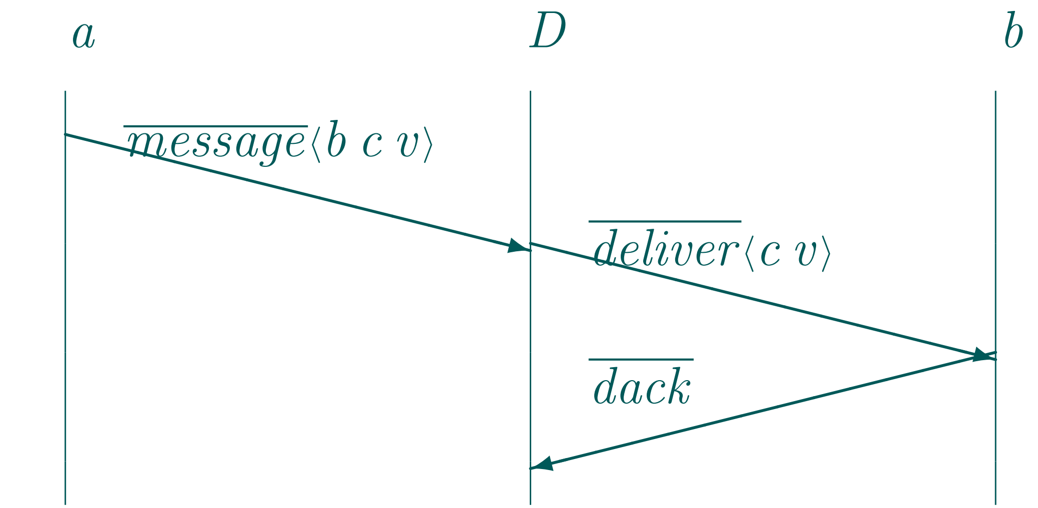
$$\longrightarrow \quad \sigma; @_a\, Q \qquad \text{if } \sigma(a) \neq \sigma(b)$$

High-Level:

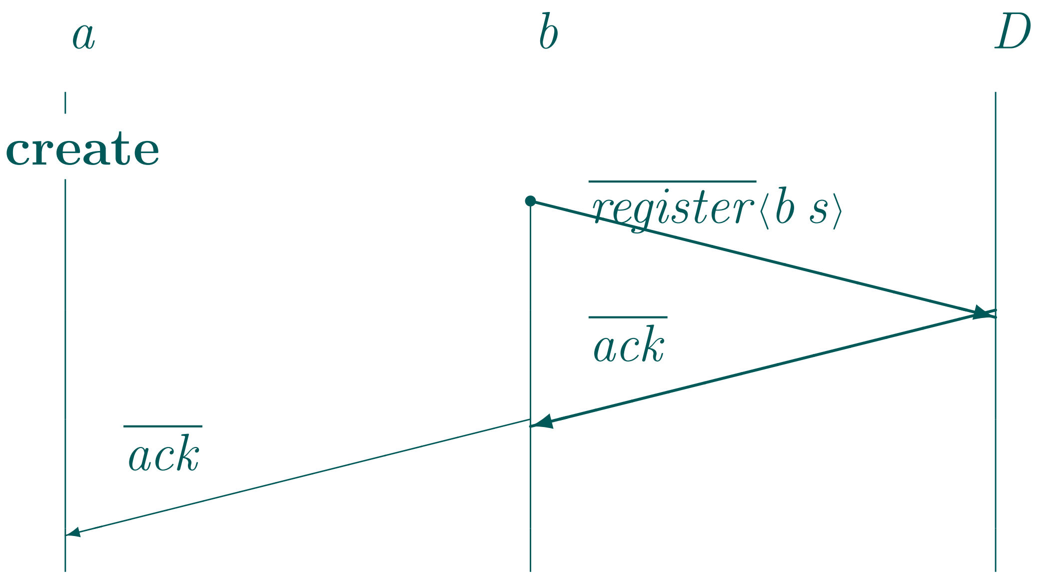
$$\sigma; @_a\, \overline{\langle b@? \rangle c}v \quad \longrightarrow \quad \sigma; @_b\, \overline{c}v$$

Example Encoding – Central Daemon

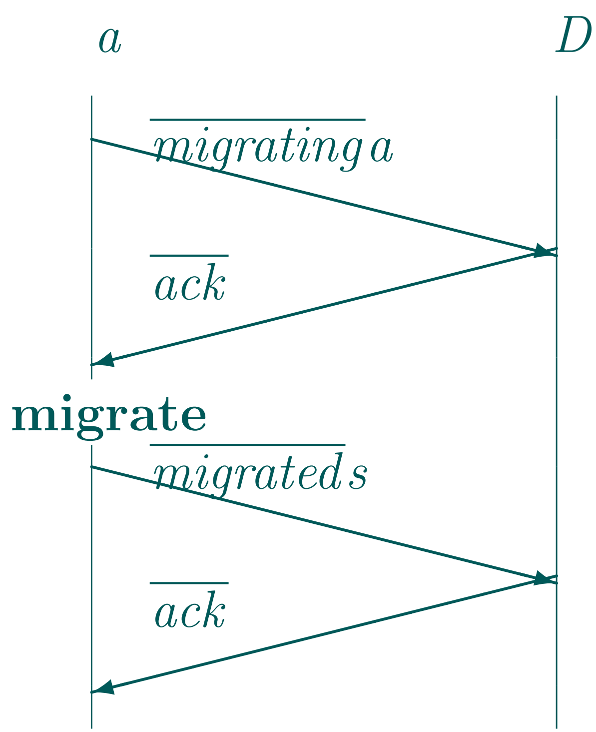
Location-independent output



Creation



Migration



Example Encoding – Central Daemon

$$
\begin{array}{lcl}
[\![\overline{\langle b@?\rangle c}v]\!]_a & = & \langle D@Dsite\rangle message![b\ c\ v] \\
[\![\textbf{agent}\ b = P\ \textbf{in}\ Q]\!]_a & = & currentlocs. \\
 & & \quad \textbf{agent}\ b = \\
 & & \qquad *deliver\, c\ v.(\langle D@Dsite\rangle dack![] \mid \overline{c}v) \\
 & & \quad\ \mid \langle D@Dsite\rangle register![b\ s] \\
 & & \quad\ \mid ack.(\langle a@s\rangle ack![] \mid \overline{currentlocs} \mid [\![P]\!]_b) \\
 & & \quad \textbf{in} \\
 & & \qquad ack.(\overline{currentlocs} \mid [\![Q]\!]_a) \\
[\![\textbf{migrate to}\ u.P]\!]_a & = & currentloc\_. \\
 & & \qquad \langle D@Dsite\rangle migrating!a \\
 & & \quad \mid ack. \\
 & & \qquad\quad \textbf{migrate to}\ u. \\
 & & \qquad\qquad \langle D@Dsite\rangle register![a\ u] \\
 & & \qquad\quad \mid ack.(\overline{currentloc}\,u \mid [\![P]\!]_a) \\
[\![0]\!]_a & = & \\
[\![P \mid Q]\!]_a & = & \\
[\![cw.P]\!]_a & = & \\
[\![*cw.P]\!]_a & = & \text{ALL HOMOMORPHIC} \\
[\![\textbf{iflocal}\ \overline{\langle b\rangle c}v.P\ \textbf{else}\ Q]\!]_a & = & \\
[\![\textbf{new}\ c\ \textbf{in}\ P]\!]_a & = & \\
[\![\textbf{if}\ a = b\ \textbf{then}\ P\ \textbf{else}\ Q]\!]_a & = &
\end{array}
$$

$$
\begin{array}{lcl}
[\![@_a\, P]\!]_s & = & @_a\,\textbf{new}\ register, migrating, message, dack, deliver, ack, currentloc\ \textbf{in} \\
 & & \quad \textbf{agent}\ D = DAEMON\ \textbf{in} \\
 & & \qquad \textbf{let}\ Dsite = s\ \textbf{in} \\
 & & \qquad\quad *deliver\, c\ v.(\langle D@Dsite\rangle dack![] \mid \overline{c}v) \\
 & & \qquad \mid \langle D@Dsite\rangle register![a\ s] \\
 & & \qquad \mid ack.(\overline{currentlocs} \mid [\![P]\!]_a)
\end{array}
$$

$$
\begin{array}{l}
DAEMON = \textbf{new}\ lock\ \textbf{in} \\
\quad \overline{lock}\,\textbf{emptymap} \\
\mid *register\, a\ s. \\
\quad lock\, m. \\
\quad \textbf{let}\ m' = (m\ \textbf{with}\ a \mapsto s)\ \textbf{in} \\
\qquad \overline{lock}\,m' \mid \langle a@s\rangle ack![] \\
\mid *migrating\, a. \\
\quad lock\, m. \\
\quad \textbf{lookup}\ a\ \textbf{in}\ m\ \textbf{with} \\
\qquad \textbf{found}(s). \\
\qquad\quad \langle a@s\rangle ack![] \\
\qquad \mid migrated\, s. \\
\qquad\qquad \textbf{let}\ m' = (m\ \textbf{with}\ a \mapsto s')\ \textbf{in} \\
\qquad\qquad \overline{lock}\,m' \mid \langle a@s'\rangle ack![] \\
\qquad \textbf{notfound}.0 \\
\mid *message\, a\ c\ v. \\
\quad lock\, m. \\
\quad \textbf{lookup}\ a\ \textbf{in}\ m\ \textbf{with} \\
\qquad \textbf{found}(s). \\
\qquad\quad \langle a@s\rangle deliver![c\ v] \\
\qquad \mid dack.\overline{lock}\,m \\
\qquad \textbf{notfound}.0
\end{array}
$$

Example Encoding – Central Daemon

$$
\begin{array}{rcl}
[\![\overline{\langle b@?\rangle c} v]\!]_a & = & \langle D@Dsite\rangle message![b\ c\ v] \\
[\![\textbf{agent}\ b = P\ \textbf{in}\ Q]\!]_a & = & currentloc\,s. \\
& & \quad \textbf{agent}\ b = \\
& & \qquad *deliver\,c\ v.(\langle D@Dsite\rangle dack![] \mid \overline{c} v) \\
& & \qquad \mid \langle D@Dsite\rangle register![b\ s] \\
& & \qquad \mid ack.(\langle a@s\rangle ack![] \mid \overline{currentloc}\,s \mid [\![P]\!]_b) \\
& & \quad \textbf{in} \\
& & \qquad ack.(\overline{currentloc}\,s \mid [\![Q]\!]_a) \\
[\![\textbf{migrate to}\ u.P]\!]_a & = & currentloc\_. \\
& & \quad \langle D@Dsite\rangle migrating!a \\
& & \quad \mid ack. \\
& & \qquad \textbf{migrate to}\ u. \\
& & \qquad\quad \langle D@Dsite\rangle register![a\ u] \\
& & \qquad\quad \mid ack.(\overline{currentloc}\,u \mid [\![P]\!]_a)
\end{array}
$$

Nomadic Pict Implementation (Wojciechowski)

Prototype implementation, and various algorithms:

`http://www.cs.put.poznan.pl/pawelw/npict.html`

Nomadic π Reasoning (Unyapoth)

Correctness of that central-server encoding:

Theorem 1.6 (roughly) For all P in the high-level calculus, $P \simeq [\![P]\!]$.

This required new observational congruences and proof techniques, e.g. to reason about the behaviour of processes that are temporarily immobile due to a lock being held elsewhere in the system.

Reflections

Our impression: good abstraction level for writing (& reasoning) such algorithms.

But for general PL design, better to go even further down:

- Many subtly different communication abstractions, even for LD – don't want to pick on one.
- Doing this as translations between calculi was good for semantics and proof (keeping it simple) but lots of work for implementation.

Hence, instead, want to be able to write them libraries for an existing language. What do we need in the language for that?

Interesting distributed abstractions have code on many sites (e.g. forwarding-pointers infrastructure daemons, more recent P2P). Hence:

- Coherence among abstract types of high-level language site names?
- Versioning!

Location-dependent dynamic rebinding – useful.

In Join and Nomadic π, a single execution of a program could become distributed

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Distributed Interaction

Non-local:

- between different invocations of a program build
- between different builds
- between different programs
- across multiple failure domains
- across multiple trust domains
- high remote/local access cost ratio

Choice of distributed abstractions

We could build in some particular communication primitives, but...

...different applications need wildly different communication infrastructure, with different synchronisation, security, and performance.

So:

1: A distributed programming language should not have any built-in network communication. Instead, have marshalling (serialization, pickling) of arbitrary values.

- this gives a level of abstraction that makes distribution explicit (so can understand failure and security issues)
- but the language needs to be expressive enough so that varied communication infrastructure can be coded as libraries.

Typed Marshalling

But: distributed programming is especially hard — so want to program that infrastructure, and applications, in a high-level type-safe language.

So:

2: a global language should provide type-safe marshalling of arbitrary values.

then can express distributed infrastructure type-safely above byte-string TCP, persistent store etc.

Underlying Theme

With distribution, we can't statically prevent all errors — but we'd like to discover them as soon as possible in the development & deployment process.

Do so by careful name generation, of runtime type and term names, so that...

...name equality testing suffices to guarantee type safety (including abstract type invariants).

Basic Type-Safe Marshalling

Machine A | Machine B

```
send(marshal 5:int)
```

```
3 + unmarshal (receive ()) as int
```

The value `5:int` is communicated.

A dynamic type equality check at unmarshal-time ensures type-safety (here `int = int` succeeds).

(here `send:string->unit` and `receive:unit->string` are from a library written in Acute, above the Sockets interface, not built-in primitives)

Basic Type-Safe Marshalling

Three strengths of dynamic check:

1. just check equality of types
2. also check the marshalled value has that type
3. just check runtime representation consistent with expected structure (Henry, Mauny, Chailloux [Hen])

(1) ok — at any type — if the marshalled value is trusted (using type system to prevent accidental errors, as usual)

(2 or 3) necessary if the marshalled value is untrusted — but in general cannot check the invariants of abstract types.

Where you can't, you shouldn't be receiving such values

Need both. For Acute we focus on (1) — the more challenging.

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Marshalling Functions – Rebinding to local resources

```
send (marshal (function x -> print_int (x+1)) :  int->unit)
```

Have to *rebind* to local `stdio` `print_int` at unmarshal-time.

(or make a distributed reference? no...)

A marshalled value might mention:

(1) ubiquitous standard library calls, e.g., `print_int`;

(2) application-specific libraries that are location-dependent, e.g. P2P routing;

(3) application code which is not location-dependent but is known to be present at all relevant sites; and

. . . .

(4) other let-bound values.

$\mapsto$ rebinding slides

Marshalling Functions – Rebinding to local resources – what to ship and what to rebind?

```
module M1  =  struct let y=6 end
··mark "MK"···································
module M2  =  struct let z=3 end
send( marshal "MK" (function ()-> (M1.y,M2.z))
                 : unit->int*int )
|
module M1  =  struct let y=6 end
module M2  =  struct let z=4 end
((unmarshal (receive ()) as unit->int*int) (), M2.z)
```

the `M1.y` reference to `M1` is rebound, whereas the first defn of `M2` is copied and sent with the marshalled value. Result `() | ((6,3),4)`.

Marshalling Functions – Rebinding to local resources – when does it take effect?

What's the relative timing of variable instantiation and (un)marshalling?

Standard CBV would substitute out all definitions – so nothing could ever be rebound. Instead use *redex-time* reduction strategy for module references: instantiate `M.x` only when it appears in redex position.

```
module M  =  struct let x=6 end
import M  :  sig val x:int end version * = M
mark "MK"
send( marshal "MK" (M.x, function ()-> M.x) : int*(unit-
>int))
```

the occurrence `M.x` is instantiated by `6` before the marshal happens, but the occurrence `M.x` would not appear in redex-position until a subsequent unmarshal and application to `()`, so it is subject to rebinding.

Rebinding to local resources – executing partial programs?

Partial programs might be written explicitly – leaving a library to be dynamically linked – or arise from unmarshalling.

1. Disallow. Have to fully link at unmarshal-time.
2. Allow. Can choose per unmarshal whether (i) to demand full linkability then or (ii) not.

(ii) permits later errors (redex-time instead of unmarshal-time). Conversely, it allows more programs to execute successfully.

For now, just (ii). Try to link an unlinked import only when a term field is needed (appears in redex position).

Rebinding to local resources – what to rebind to?

Add *resolvespec* data to imports, for example:

```
import M : sig val y:int end
  by "http://www.acute.org/M"  = unlinked
M.y + 3
```

Should the *resolvespec* language be

1. general (Turing complete), or
2. restricted?

(1) sometimes necessary. (2) allows analysis of an upper bound on the set of modules a program may demand (cf disconnection).

For now, have a list of URIs and `Here_already`.

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Local type equalities in ML module systems

In ML type fields can either be *abstract*, with the representation type held private to the module body, or *concrete*, with the type field in the signature manifestly equal to its representation type.

```
module  Mabstract
 :    sig   type t      val get:t->int            ... end
 = struct   type t=int  let get=function x->x     ... end

module  Mconcrete
 :    sig   type t=int  val get:t->int            ... end
 = struct   type t=int  let get=function x->x     ... end
```

In this scope $\vdash$ `Mconcrete.t` $=$ `int`.

(c.f. Harper & Pierce, ATTAPL)

Marshalling for Abstract Types – The Problem

```
module BalancedTree
 : sig                                      = struct
     type t                                     type t = int tree
     val empty : t                              let empty = ...
     val insert : int -> t -> t                 let insert i x = ...
     ...                                        ...
   end                                        end ;;
send ( marshal e : BalancedTree.t )
```

What dynamic type check should we do at unmarshal-time?

Want not just type safety with respect to the representation type, but also *abstraction safety*, i.e. the receiver should respect the invariants of `BalancedTree.t`.

Solution: construct *globally meaningful runtime type names*.

Marshalling for Abstract Types – Summary of Main Cases

Interface	Implementation	Desired behavior
same	same code; effect-free	√ succeed
same	same internal invariants	? maybe
same	same external behaviour but different internal invariants	✗ fail
same	different external behaviour	✗ fail
different	...	✗ fail
...	different representation types	✗ fail

Naming: global type names (1 of 3)

Case 1: For effect-free modules, construct names by *hashing* module definitions.

(taking their dependencies properly into account...)

For example, the runtime type name for `BalancedTree.t` is h`.t`, where the hash h is roughly

```
hash(
  module BalancedTree
   : sig                              = struct
       type t                            type t = int tree
       val empty : t                     let empty = ...
       val insert : int -> t -> t        let insert i x = ...
       ...                               ...
     end                              end ;;
)
```

Naming: global type names (2 of 3)

Case 2: For effect-full modules, e.g.

```
module fresh NCounter
: sig                  =  struct
    type t                  type t=int
    val start:t             let start = 0
    val get:t->int          let get = fun (x:int)->x
    val up:t->t             let up =
                              let step=IO.read_int() in
                              fun (x:int)->step+x
  end                     end
```

construct names freshly at run-time.

Implementation: hashes and fresh names are all 160-bit numbers. Fresh names are generated randomly.

Naming: global type names (3 of 3)

Case 3: ...or, for effect-free modules, allow the programmer to force compile-time generation of a fresh name, thus allowing shared types between distributed programs that *link* against the *same object file*.

Marshalling for Abstract Types – Technicalities

- type system based on singleton kinds [Harper et al, Leroy]
- add `h.t` to type grammar, where h is either a hash or a fresh name
- type rules check `h.t` used correctly, but the implementation never needs to look inside a hash
- compilation (or module initialisation):
  - constructs a hash or name h for each module
  - selfifies signatures, replacing `type t` by `type t=h.t`
  - normalises types, replacing `M.t` by either `h.t` (if abstract) or by the manifest `T`
  - (have to deal with references to earlier type fields in a module)
- these normalised types can be compared with *syntactic equality* at unmarshal time
- for sanity, the runtime semantics uses *coloured brackets* $[e]^{T}_{eqs}$ GMZ
- avoid recursive hashes

Marshalling for Abstract Types – Breaking Abstractions

In the presence of version change, sometimes need to break earlier abstractions – e.g. to provide a new version of an abstract type, type-compatible with the old.

The new version should satisfy the same key invariants, but may have a bug fix, performance fix, or extra functionality.

Turing doesn't let us check "same key invariants" (and typically they have never been expressed precisely), so we let the programmer assert it.

For example

```
module BalancedTree'
  : sig                                      = struct
      type t = BalancedTree.t                    type t = int tree
      val empty : t                              let empty = ...
      val insert : int -> t -> t                 let insert i x = ...
      ...                                        ...
    end                                        end

with! BalancedTree.t = int tree
```

Choices:

- use the extra equation in the module body, or just at the interface?
- have to specify the representation type explicitly?

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Naming: establishing shared, typed, expression-level names

Need shared, typed, names for distributed channels, RPC handles, etc.

Can use the same machinery to build values of FreshML *t* `name` types:

Suppose we have a module `DChan` which implements a distributed `DChan.send` by sending a marshalled pair of a channel name and a value across the network.

```
module hash DChan :
  sig
    val send : forall t. t name * t -> unit
    val recv : forall t. t name * (t -> unit) -> unit
  end
```

How to establish a name shared between sender and receiver code such that testing name equality ensures type correctness of communication?

Naming: establishing shared, typed, expression-level names

Scenario 1: Sender and receiver both arise from a single execution of a single build of a single program. Use expression-level runtime `fresh`.
(the JoCaml and Nomadic Pict semantics)

Scenario 2: Sender and receiver in different programs, but both are statically linked to a structure of names that was built previously. Use expression-level compile-time `cfresh`.
(a typed form of off-line GUID generators)

Scenario 3: Sender and receiver in different programs, but both share the source code of a module `M` that defines the RPC function `f` used by the receiver. Use `hash(M.f)`.
(just works, without prior exchange of names at build- or run-time)

Scenario 4: Sender and receiver in different programs, sharing no source code except a type and a string. Use `hash(int,"foo")`. (minimum shared information – a typed form of "traders")

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Version Change

We don't just have pervasive distributed execution, but also:

- distributed software development and deployment,
- decentralized over many different administrative domains,
- on long timescales.

Cannot synchronize software updates, so:

3. We must support interaction between different executions and different versions of programs.

- How can we retain type safety and abstraction now?

In particular, need globally coherent notion of type equality in the presence of version change.

Versioning

Good Old-Fashioned Software: (mostly) ensure coherent set of modules at build time, with a single source tree, CVS, `make`, etc.

Global Software Development: dynamic linking and rebinding. Need versions and version constraints.
(programmer-specified approximations to behavioural specs)

First cut: take some arbitrary languages of version numbers vn, constraints vc, and satisfaction $vn \in vc$.

```
module M version 2.3.5 = struct let y=22  end

import M version 2.3.* : sig    val y:int end
M.y + 3
```

Check $vn \in vc$ at compile-time and at dynamic-link-time (in addition to signature matching). Meaning of versions left to social process...

Versioning – Balance of Power

Sometimes need tighter version control – to ensure that only a mutually tested collection of modules can interact out there.

Can use hash machinery: insisting on *exact matching of module hashes* gives an analogue of GOFS – use that for default marshalling behaviour.

Choose whether code producer or code consumer has control.

Subtle interactions between versions, hashes, and type equality...

Versioning – Expressiveness

Should the version satisfaction relation be

1. built-in, and simple (as above), or
2. arbitrary code (Turing complete), with modules parametric on types of versions and constraints?

(just as for *resolvespecs*) Again (1) allows static analysis of what linking will succeed, but we may not wish to prescribe a single all-encompassing version scheme.

Should versions contain hereditary data?

Interactions: Rebinding, Type Abstraction, Versions

Without rebinding, module references `M.t` and `M.x` are *definite*.

```
module M : sig    val f:int->int         end
         = struct let f=function x->x+2 end
module EvenCounter
: sig                  = struct
    type t                 type t=int
    val start:t            let start = 0
    val get:t->int         let get = function x->x
    val up:t->t            let up=function x->M.f x
  end                    end
```

The body of `EvenCounter` uses a unique `M`. Hashing follows this: the hash of `EvenCounter` mentions h`.f`, where h is the hash of `M`. Marshalling of `EvenCounter.t` values is abstraction-safe.

Interactions: Rebinding, Type Abstraction, Versions

But... rebindable module references are *indefinite*.

```
module M : sig    val f:int->int          end
         = struct let f=function x->x+2 end
import M : sig val f:int->int end  version * = M
mark "MK"
module EvenCounter : ... = ...M.f...
send( marshal "MK"  (function () -> EvenCounter.get
   (EvenCounter.up EvenCounter.start)):unit->int)
|
module M : ... = struct let f=function x->x+3 end
(unmarshal (receive ()) as unit->int) ()
```

Typically want a tighter version constraint in place of `*` – e.g. `2.3.*` or even an exact-name constraint.

Interactions: Rebinding, Type Abstraction, Versions

Then... what should the global type name for `EvenCounter.t` be?

Can rebind to any `M` matching

```
import M : sig val f:int->int end  version 2.3.*
```

so in building a hash of `EvenCounter` should replace `M.f` by h`.f` where h is the hash *of that import*.

The hash of an import is a global name for the set of modules that can be bound to it.

Interactions: Rebinding, Type Abstraction, Versions

But to make that sound we need to constrain the set of modules that imports of modules with abstract types can be linked *to*, to ensure they all have the same representation type.

Add a *likespec* to such imports, e.g.

```
import M : sig type t  val x:t  end
        version 2.3.*
        like  struct type t=int end
```

or more typically

```
import Graphics : GraphicsSig
               version 2.3.*
               like Graphics2_0
```

Interactions: Marshalling within abstraction boundaries

```
module EvenCounter
: sig
    type t
    val start:t
    val get:t->int
    val up:t->t
    val send : t -> unit
    val recv : unit -> t
  end
= struct
    type t=int
    ...
    let send = fun (x:t) -> IO.send(marshal "StdLib" x : t)
    let recv = fun () -> (unmarshal(IO.receive()) as t)
  end
```

Back to computation mobility

If we can marshal arbitrary values...

...then to support computation mobility (as in DJoin/Nomadic Pict etc) it suffices to turn computations into values.

Computation mobility via thread thunkification

Atomically convert a collection of threads, mutexes, and cvars, to a thunk. Those thunks can be marshalled, just like any other value.

```
let rec delay x = if x=0 then () else delay (x-1) in
let rec f x = IO.print_int x; IO.print_newline (); f (x+1) in
let t1 = fresh in
let _ = create_thread t1 f 0 in
let _ = delay 15 in
let v = thunkify ((Thread (t1,Blocking))::[]) in
IO.send( marshal "StdLib" v : thunkkey list -> unit )
```

—

```
let rec delay x = if x=0 then () else delay (x-1) in
let exit_soon = create_thread fresh
  (fun () -> delay 15 ; exit 0) () in
let v = (unmarshal(IO.receive()) as thunkkey list -> unit) in
v ((Thread (fresh,Blocking))::[])
```

Computation mobility via thread thunkification

Atomically convert a collection of threads, mutexes, and cvars, to a thunk. Those thunks can be marshalled, just like any other value.

Analogous to `call/cc` — but to a boundary further out than usual: capturing a parallel evaluation context (comprising a set of named threads/mutexes/cvars) and removing it from the executing scheduler.

Thunkify: Interactions

- thread naming (how unique? provided by runtime or programmer?)
- references, names, marshalling, and thunkify

  treat locations and names differently: marshalling locations involves a deep copy; marshalling names just gives the names; thunkify of threads/mutexes/cvars is destructive
- module initialisation, concurrency, and thunkify

  second-class module system — so can't thunkify a thread that is executing module initialisation
- thunkify vs inter-thread synchronisation primitives (key!)
  - mutex/cvar primitives
  - OS blocking calls

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

Acute

We set out in 2003–5 to build a prototype language, Acute (with a Caml core), to experiment with these ideas, building on our earlier calculi and (in the end) going well beyond them.

1. exploration of design space
2. complete Acute language definition (types, compilation, operational semantics, 80pp)
3. Acute implementation (runtime interprets AST+closures, 25kloc FreshOCaml)

Docs, source and binary distros (i386-linux & MacOSX) at `http://www.cl.cam.ac.uk/users/pes20/acute/`. Try it!

Good for non-trivial examples, but not intended as a production language.

No proofs, but... implementation can do per-step runtime type checking.

Acute Summary – Constructs

Mostly-conventional ML core and:

$T ::= ... | T\ \mathsf{name} | \mathsf{thread} | h.\mathrm{t} | \mathrm{n}$

$e ::= ... | \mathbf{marshal}\ e_1\ e_2 : T | \mathbf{unmarshal}\ e\ \textbf{as}\ T |$
$\mathbf{fresh}_T | \mathbf{cfresh}_T | \mathbf{hash}(\mathrm{M}_M.\mathrm{x})_T | \mathbf{hash}(T, e_2)_{T'} | \mathbf{hash}(T, e_2, e_1)_{T'} |$
$\mathbf{swap}\ e_1\ \mathbf{and}\ e_2\ \mathbf{in}\ e_3 | \mathbf{support}_T\, e | \mathbf{thunkify} | [e]^T_{eqs}$

$sourcedefinition ::=$
$\mathbf{module}\ mode\ \mathrm{M}_M : Sig\ \mathbf{version}\ vne = Str\ withspec\ |$
$\mathbf{import}\ mode\ \mathrm{M}_M : Sig\ \mathbf{version}\ vce\ likespec\ \mathbf{by}\ resolvespec = Mo |$
$\mathbf{mark}$ "MK"

$T ::= \mathsf{int} | \mathsf{bool} | \mathsf{string} | \mathsf{unit} | \mathsf{char} | \mathsf{void} | T_1 * .. * T_n | T_1 + .. + T_n | T \rightarrow T' | T\ \mathsf{list} | T\ \mathsf{option} | T\ \mathsf{ref} | \mathsf{exn} |$
$\mathrm{M}_M.\mathrm{t} | t | \forall\, t.T | \exists\, t.T | T\ \mathsf{name} | T\ \mathsf{tie} | \mathsf{thread} | \mathsf{mutex} | \mathsf{cvar} | \mathsf{thunkifymode} | \mathsf{thunkkey} | \mathsf{thunklet} | h.\mathrm{t} | \mathrm{n}$

Examples

Think this suffices for typeful programming of multi-layered, distributed, evolving systems.

Examples – libraries for:

- Minesweeper game. Marshals game state to persistent store to save.
- RFI/Distributed channels/local channels/TCP string messaging/TCP connection managment
- Nomadic Pict. Mobile computations that can be migrated between machines, with distributed asynchronous messaging.
- Bounce (above Nomadic Pict library).
- Ambient primitives. Tree-structured mobile computations.

Moderate-scale — around 1000 lines each. Weeks, not years.

Examples: The Ambient API

```
module hash! Ambients :
    sig
      val ambient : string -> (unit -> unit) -> unit
      val spawn : (unit -> unit) -> unit
      val c_in : string -> unit
      val c_out : string -> unit
      val c_open : string -> unit

      val init : (Tcp.ip option * Tcp.port option) -> unit
      val migrate : Tcp.addr -> unit
    end
    =
...
```

Examples: The Npi API

```
module hash! Npi :
 sig
   type group
   val create_group : forall t. (t -> unit) -> t -> unit
   val create_gthread : forall t.  (t->unit) -> t -> unit
   val recv_local : forall t. t name -> t
   val send_local : forall t. t name -> t -> unit

   val init : (Tcp.ip option * Tcp.port option) -> unit
   val send_remote : forall t.  string
            -> (Tcp.addr * group name * t name) -> t  -> unit
   val migrate_group : Tcp.addr -> unit
   val local_addr : unit -> Tcp.ip option * Tcp.port
 end
```

Demo

In `examples`:

```
./go6 minesweep.ac
```

```
./go6 npi-recv.ac
./go6 bounce.ac
```

Look at code for the latter, and `npi.ac`

See also `examples/ambients` (ambient-primitives library, and a little OCaml preprocessor to provide calculus-like syntax).

For small examples it can be interesting to turn on compiler flags to print hash types etc.

Reflections

- basic ideas pretty solid (typed marshalling, module-level redex time reduction and rebinding, fresh and hash run-time type names for abstract types, versioning at module level, thunkify)

  (other things more debatable or just not addressed – will return to these)

- keeping impl and defn in sync was essential

- doing (semi-)full-scale language design was essential – many unsuspected interactions between features

  (warning: can be tricky to get this kind of thing across in 12 pages)

- ...but, all this is a lot of engineering – and the resulting impl only good for modest examples.

- So, one more time, with feeling...

- Local concurrency: π calculus (92) and Pict (95) background
- Mobile computations: Join (96) and Nomadic Pict (98)
- Marshalling: choice of distributed abstractions, and trust assumptions
- Dynamic rebinding and evaluation strategies
- Type equality between programs: run-time type names
- Typed interaction handles: expression-level names
- Version change — and interactions between language features
- Acute: semantics and implementation
- HashCaml: type- and abstraction-safe distribution for OCaml

One more time, with feeling...

HashCaml: Type-Safe Distributed Programming for OCaml.

OCaml has marshalling of arbitrary values — but

> *"Anything can happen at run-time if the object in the file does not belong to the given type."*

We adapt the OCaml bytecode compiler and runtime to support type-safe and abstraction-safe marshalling, and associated expression-level name constructs.

Scope

From Acute:

- include fresh and hash-generated runtime type and term names.
- omit dynamic rebinding (except to the standard library), dynamic linking, explicit versioning, thunkify
- use OCaml marshaller under the hood – so marshalling of function values only permitted between identical builds

From OCaml:

- keep the existing OCaml static type system (except where really forced)
- don't address marshalling for objects, polymorphic variants, extensions
- aim to address all the rest (get pretty close)

Main issues

- the static and dynamic type equalities for features in (OCaml-Acute): variant and record types, substructures, arbitrary ascription, functors, separate compilation, external functions
- marshalling within polymorphic functions and the value restriction
- implementation strategy

Throughout: pragmatic issues with existing OCaml language design and implementation.

Ascription — the simple case

Abstract types in OCaml (as in SML) are introduced by signature ascription, so statically:

```
module M
= struct
    type t = int
    let x = 3
  end
module M1 : sig type t  val x:t end = M
let id : M.t -> M1.t = fun z->z   (* does not typecheck *)
```

(Acute was simpler: there each `struct` had a unique associated `sig` — but so far this seems identical...).

Ascription — the simple case

We mirror that in the HashCaml dynamic type equality:

```
module M
= struct
    type t = int
    let x = 3
  end
module M1 : sig type t  val x:t end = M
let s = Marshal.to_string M.x []
let (z : M1.t) = Marshal.from_string s 0
                 (* raises Unmarshal exception *)
```

Ascription — the fresh case

```
module NCounter
: sig                      = fresh struct
    type t                     type t=int
    val start:t                let start = 0
    val get:t->int             let get = fun (x:int)->x
    val up:t->t                let up =
  end                            let step = read_int () in
                                 fun (x:int)->step+x
                           end
```

Allow hash/fresh semantics arbitrarily, as needed hash semantics for effectful modules in NPi and Ambient examples. Could base default on valuability analysis.

The `myname` Implementation Strategy

Where do we keep these runtime type names?

In the OCaml implementation, structures are compiled to records, and ascriptions, e.g. `module M : Sig = M'`, have no existence at runtime (except for coercion functions).

So we add a secret `myname` field to each structure, containing its hash or fresh/cfresh name (h).

For usages of the runtime type name of an abstract `M.t`, we generate lambda code to build `(M.myname, "t")`.

But...

note that does give us a discrepancy between static and dynamic type equalities:

```
module M = struct type t = int  let x = 3 end
module M1 : sig type t val x:t end = M
module M2 : sig type t val x:t end = M
let id : M1.t -> M2.t = fun x -> x        (* statically fails *)
```

```
module M = struct type t = int  let x = 3 end
module M1 : sig type t val x:t end = M
module M2 : sig type t val x:t end = M
let s = Marshal.to_string M1.x []
let (z : M2.t) = Marshal.from_string s 0  (* dynamically ok *)
```

Does this matter? If so, which should we fix...?

Can dynamically compare types *across* static scopes

```
module M1
: sig
    module M2 : sig type t  val x:t end
  end
= struct
    module M2 : sig type t  val x:t  val f:t->t end
    = struct
        type t = int
        let  x = 2
        let  f = fun z->z+2
        let  _ = (s3 := Marshal.to_string (3 : t) [])
      end
      let _=(s2:=Marshal.to_string(M2.f M2.x : M2.t) [])
  end;;
let _ = (s1:=Marshal.to_string (M1.M2.x : M1.M2.t) [])
```

Prefix Hashing

When we build the hash of a module, we have to take account of its dependencies.

But how much?

```
module M
= struct
    let iterate = List.iter
    let x = 7
    module N = struct  let y = x  end
    module O = struct  let w = 4  end
  end
```

Depend on all the superstructure prefix? Regular but very awkward in practice, e.g. making the result of a use of `Set.Make` in a file depend on all the preceeding definitions.

Applicative Functors

OCaml functors are all statically applicative, so in the scope of

```
module F (X:sig end) : sig type t end
                     = struct type t=int end
module U = struct end
module M = F(U)
module M'= F(U);;

((fun x->x) : M.t -> F(U).t);;
((fun x->x) : M.t -> M'.t);;
```

we have static type equalities `M.t = M'.t = F(U).t`. We mirror that in the dynamic equality by constructing the hashes of `M` and `M'` functionally.

Common cases, e.g. marshalling a value of a set or hashtable abstract type produced by applying an OCaml standard library functor, should therefore just work. (currently we reevaluate... sometimes wrong)

Abstract Type Operators are Applicative

in both static and dynamic equalities — the latter again by functional construction of hashes:

```
module M
= struct
    type 'a t = int * 'a
    let f x = (3,x)
  end
module M1 : sig type 'a t  val f:'a -> 'a t end = M
let s = Marshal.to_string (M1.f true) []
let (z : bool M1.t) = Marshal.from_string s 0
```

Here the runtime type name for `bool M1.t` is essentially `((M1.myname,"t"),[bool])`.

Variant Types

```
module M1 = struct  type t = C of int  end
module M2 = struct  type t = C of int  end
let f : M1.t -> M2.t = fun x->x  (* statically fails *)
```

Conceivable dynamic equalities: build runtime type reps

- freshly at runtime (ok for single runs — otherwise useless)
- from a hash of the type definition and its path (demands too much similarity?)
- from a hash just of the type definition
- from a hash of a normalised definition, ignoring the order of clauses
- from a hash of the structure of the definition, ignoring the names of constructors, or
- any of the above, chosen per-type by the programmer. (insane?)

Separate Compilation

just works ok

(the plumbing is handled automatically, due to the `myname` implementation strategy.

On-the-wire Marshalled Type Representations

In marshalled values: just 256-bit numbers.

Only operation on these is equality at unmarshal time

(but for subtyping [Deniélou, Leifer; ICFP06] or intensional type analysis would need more structure)

Internal Runtime Type Representations

Must build them compositionally in the type structure:

```
let pair_and_marshal : 'a -> string
  = function x -> Marshal.to_string (x,x) []
let s = pair_and_marshal 17
let n = let (x1,x2)=(Marshal.from_string s 0) in x1+x2
```

and want to avoid re-hashing all over — so use a good representation internally.

Implementation builds a type-passing translation, adding type parameters at every generalization point. Conceptually naive (though not simple to actually do!), but performance not too shabby.

Marshalling at Non-Ground Runtime Types

Wouldn't be in keeping with SML/OCaml ML polymorphism (and would need structured type representations). So don't.

```
let x = []
let s = Marshal.to_string x [] (* fails *)

let f x =
  let s = Marshal.to_string x [] in  (* succeeds *)
  x + 1 in
f 0

let x=function y -> y
let s=Marshal.to_string (x:int->int) [Marshal.Closures]
                                              (* succeeds *)
```

The Value Restriction

SML97 adopted the *value restriction* [Wri95], allowing polymorphic generalisation only for syntactic values. In OCaml 3.09.1 generalization is more liberal, in three ways:

- whether a conditional expression is generalizable is independent of the condition;
- certain application expressions are generalizable; and
- type variables that occur only on the right of an arrow can be generalised, as proposed by Garrigue [Gar04].

All three would lead to problems with our type-passing translation, as inserting the internal type-representation lambdas would change the evaluation order (twisty examples in the paper). Hence, HashCaml imposes the value restriction.

Implementation Intricacies

Lots

Expression-level name features

There is a new family of types `'a name`, represented as 256-bit values. Expression-level names can be generated in several ways:

(1) freshly: just write `fresh` (of type `'a name`);

(2) from a pair of a type rep and a string, writing `hashname(`*t*`,` *s*`)`, where *t* is a type and *s* is a string, to yield a value of type *t* `name`;

(3) using the module hashes produced for calculating type representations, for example `fieldname M.f` (of type *t* `name` where `M.f:`*t*); and

(4) by applying name coercions `namecoercion(`*path1*`,`*path2*`,`*e*`)`

There is also a special conditional for comparing names:

`ifname` *e1*`=`*e2* `then` *e3* `else` *e4*

where *e1* `:` *t1* `name`, *e2* `:` *t2* `name`, and and *e3* is typechecked in an environment where they are unified.

Example

Source distro: `hashcaml-3.09.1-alpha-795`.

`http://www.cl.cam.ac.uk/~pes20/hashcaml/index.html`. Try it!

See the DCL example therein.

Reflections

- This is a pretty good implementation — try it and see! But it would need more engineering to make really robust, and some additional features would be very handy (existentials, bytecode marshalling, thunkify, native-code implementation).
- Type-safe marshalling isn't a big syntax or static type system change — but the dynamic type equality cuts across most of the existing language. The static and dynamic type equality should be designed together, from the start.
- At several places the existing OCaml (or SML) static equality wouldn't be the most useful dynamic equality:
  - variant and record types
  - complex ascriptions
    (unclear how much of the current expressiveness is used?)
  - value restriction

Summing up

Seen an arc from simple calculi to almost-real language design.

Understood some problems — but many still left open, even in isolation, let alone integrated into a whole.

The End

(this is only a very partial list — Google for the others...)

References

[AG97] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security, Zürich*, pages 36–47. ACM Press, April 1997.

[AP94] R. M. Amadio and S. Prasad. Localities and failures. In P. S. Thiagarajan, editor, *Proceedings of* 14^{th} *FSTTCS. LNCS 880*, pages 205–216. Springer-Verlag, 1994.

[AVWW96] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. 2nd ed.

[BHS$^+$03] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Proc. ICFP*, 2003.

[BRS$^+$05] D. Le Botlan, A. Rossberg, C. Schulte, G. Smolka, and G. Tack, 2005. `www.ps.uni-sb.de/alice/`.

[BSSs06] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *Proc. ML06, ACM SIGPLAN Workshop on ML*, September 2006. To appear.

[Car95] L. Cardelli. A language with distributed scope. In *Proc. 22nd POPL*, pages 286–297, 1995.

[CG98] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), ETAPS'98, LNCS 1378*, pages 140–155, March 1998.

[FGL$^+$96] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR '96. LNCS 1119*, pages 406–421. Springer-Verlag, August 1996.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming, Nara, LNCS 2998*, April 2004.

[Hen] Grégoire Henry. Type-safe unmarshalling for objective caml, `http://www.pps.jussieu.fr/~henry/marshal/`.

[JoC03] JoCaml. `http://moscova.inria.fr/jocaml/`, 2003.

[Kna95] F. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, December 1995.

[LPSW03] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global abstraction-safe marshalling with hash types. In *Proc. 8th ICFP*, 2003.

[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*. 2000.

[Rep99] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[RH99] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of the 26th POPL*, San Antonio, January 1999. ACM Press.

[Sew98] Peter Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *Proceedings of ICALP '98. LNCS 1443*, pages 695–706. Springer-Verlag, July 1998. See also Technical Report 435, University of Cambridge, 1997.

[Sew01] P. Sewell. Modules, abstract types, and distributed versioning. In *Proc. 28th POPL*, 2001.

[SLW$^+$04] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, Francesco Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Design rationale and language definition. Technical Report 605, University of Cambridge Computer Laboratory, October 2004. Also published as INRIA RR-5329. 193pp.

[SLW$^+$05] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. In *Proceedings of ICFP 2005: International Conference on Functional Programming (Tallinn)*, September 2005.

[SLW$^+$06] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation, 2006.

[SV99] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop. Mordano, Italy*, pages 136–150. IEEE Computer Society, June 1999.

[SV00] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proceedings of CSFW 00: The 13th IEEE Computer Security Foundations Workshop.*, pages 269–284. IEEE Computer Society, July 2000.

[SWP98a] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Proceedings of the Workshop on Internet Programming Languages (Chicago)*, May 1998.

[SWP98b] Peter Sewell, Paweł T. Wojciechowski, and Benjamin C. Pierce. Location independence for mobile agents. In *Workshop on Internet Programming Languages, Chicago*. Springer-Verlag, May 1998. Full version as Technical Report 462, University of Cambridge, and in LNCS 1686.

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

[TLK96] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *CONCUR'96, LNCS 1119*, 1996.

[US01] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proc. POPL*, pages 116–127, January 2001.

[VC98] Jan Vitek and Guiseppe Castagna. Towards a calculus of mobile computations. In *Workshop on Internet Programming Languages, Chicago*, May 1998. Full version in LNCS 1686.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

[WS99] Paweł T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and infrastructure design for mobile agents. In *Proceedings of ASA/MA 99: Agent Systems and Applications/Mobile Agents (Palm Springs)*, pages 2–12, October 1999. Best paper award.