On Implementations and Semantics of a Concurrent Programming Language

Peter Sewell[1]

Abstract

The concurrency theory literature contains many proposals for models of process algebras. We consider an example application of the π-calculus, the programming language Pict of Pierce and Turner, primarily in order to see how far it is possible to argue, from facts about the application, that some model is the most appropriate. We discuss informally the sense in which the semantics of Pict relates to the behaviour of actual implementations. Based on this we give an operational model of the interactions between a Pict implementation (considered as the abstract behaviour of a C program) and its environment (modelling an operating system and user). We then give a class of abstract machines and a definition of abstract machine correctness, using an adapted notion of testing, and prove that a sample abstract machine is indeed correct. We briefly discuss the standard of correctness appropriate for program transformations and the induced precongruence. Many of the semantic choices do indeed turn out to be determined by facts about Pict.

1 Introduction

The concurrency theory literature contains many proposals for models of process algebras, as can be seen for example from the surveys by van Glabeek [Gla90, Gla93] of certain models that are quotients of labelled transition systems (LTS's). This diversity poses a problem: for any particular application of a process algebra how can an appropriate model be selected? In this paper the problem is addressed for an example application of the π-calculus of Milner, Parrow and Walker [MPW92], primarily in order to see how far it is possible to argue, from facts about the application, that some model is the most appropriate. The application is the Pict programming language, based on the π-calculus, of Pierce and Turner [PT97]. We consider models that are quotients of the terms by congruence relations defined using notions of observation. The core of the paper is devoted to defining a notion of observation that can be seen to be appropriate for Pict. We discuss in detail the interactions between an actual Pict implementation and a user, together with their relationship to the structured operational semantics (SOS). This discussion is, in the absence of a semantics for the implementation language, necessarily informal. We then incorporate a number of simple but essential facts about the interactions into a formal model, giving precise definitions of a class of abstract machines and of a notion of observation suitable for defining abstract machine correctness. These are used to define an appropriate observational precongruence. We prove that a sample abstract machine is indeed correct and give some characterisation results, relating the observational preorder and precongruence to standard notions of testing and bisimulation.

[1]Computer Laboratory, University of Cambridge. Email: `Peter.Sewell@cl.cam.ac.uk`

Some of the discussion and technical work is necessarily specific to Pict. Much, however, should be applicable to other concurrent programming languages that do not prescribe a particular implementation scheduling strategy, for example Facile [TLK96], CML [Rep92], Concurrent Haskell [JGF96], and the Join calculus [FG96].

To define what is a correct abstract machine one must specify the required relationship between the LTS semantics of programs, as given by a π-calculus structured operational semantics, and their behaviour when executed. It is thus an essential part of the language definition, together with definitions of the syntax, type system, SOS and libraries. It must satisfy three rather pragmatic criteria. Firstly, it must be strong enough to give programmers sufficient guarantees about the behaviour of programs. Secondly, it must be loose enough to admit any 'reasonable' implementation and 'reasonable' compiler optimisations. Thirdly, it must be sufficiently mathematically tractable to allow correctness proofs for abstract machines and program transformations to be carried out. Our approach to defining abstract machine correctness is as follows. In §2 we introduce Pict and discuss informally the relationship between the SOS and implementation behaviour. The current implementation compiles a Pict program to a C program which is then compiled and executed. In §3 we give an operational model of the interaction between such a C program and the operating system which forms its immediate environment. We then give an analogous model of the interaction between the LTS semantics of programs given by the SOS and an environment. In §4 we relate the two, defining abstract machine correctness via an adaptation of the *testing preorder* of De Nicola and Hennessy [DH84] (we show that the standard notion is inappropriate). We also give a sample abstract machine, closely based on the current implementation, and prove it correct. Finally in §5 we consider program transformation, defining an observational preorder, giving some examples and proving that the induced precongruence is refined by a simple notion of bisimulation.

There is an extensive theoretical literature discussing behavioural equivalences of process calculi that are induced by some kind of tests or observations of processes, e.g. [HM80, Mil81, DH84, Hoa85, Abr87, AV93, Gla90, Gla93, San93]. The argument that the testing scenario we adopt is appropriate for Pict relies on some essential differences between Pict and any process calculus considered only in the abstract. Firstly, Pict is a *programming language*, with a fixed interpretation of nondeterminism (as a loose specification of the required implementation behaviour). It is not a modelling or simulation language, which would fix other interpretations, or a pure process calculus, which would not be committed to any interpretation. Secondly, Pict is *implemented and used.* There are clear intuitions for the intended use of the language and the behaviour of 'reasonable' implementations that can be appealed to. Further, there are libraries providing specific primitives by which a Pict program can interact with its immediate environment. The behaviour of these primitives in any reasonable implementation is well understood, giving us a solid foundation upon which to base our formal model and argue for its accuracy.

2 Pict: SOS and Implementation

Pict has a rich type system and high level syntax. This syntax is translated (in both the semantics and the implementation) into a core syntax which is a mild extension of an asynchronous choice-free π-calculus. For discussion of the design decisions underlying Pict, and of the implementation issues, we refer the reader to [PT97] and [Tur96]. For this paper we are largely concerned with the behaviour of whole programs. These interact with their environment only by communicating on channels (provided by the libraries) of rather simple types. We work with an idealized Pict, taking only these rather simple types and only the core syntax. In fact, we omit also nested tuples, records and polymorphic packages from the core, and add an equality test at base types (to replace library routines providing case analysis). These idealizations should not significantly affect the behaviours expressible by whole programs.

We take an infinite set $\mathcal{X}$, of *names*, with $\tau \notin \mathcal{X}$, and a set $\mathcal{T}$, of *base types* provided by the libraries (including e.g. the naturals), ranged over by t. The *types* are given by $T ::= t \mid !\langle T_1, \ldots, T_n\rangle \mid ?\langle T_1, \ldots, T_n\rangle \mid \updownarrow\langle T_1, \ldots, T_n\rangle$. The latter three are the types of names (or channels) along which a program can respectively output, input, and output or input tuples of names, of types $T_1, \ldots, T_n$. We write $\vec{T}$ for a tuple $T_1, \ldots, T_n$. We order types by $\leq$, which is the least preorder such that $\updownarrow\vec{T} \leq\, !\vec{T}$ and $\updownarrow\vec{T} \leq ?\vec{T}$. This is simply a notational convenience, we will not have substantive subtyping. *Type contexts* Γ are partial functions from $\mathcal{X}$ to types with $\mathcal{X} - \mathrm{dom}(\Gamma)$ infinite. We write Γ, Δ for the union of partial functions Γ and Δ with disjoint domains and $\mathcal{X} - \mathrm{dom}(\Gamma) - \mathrm{dom}(\Delta)$ infinite. *Process terms* are

$$P \quad ::= \quad 0 \;\Big|\; \overline{x}\langle\vec{z}\rangle \;\Big|\; x(\vec{y}).P \;\Big|\; !\,x(\vec{y}).P \;\Big|\; P\,|\,P \;\Big|\; (\nu y\!:\!T)P \;\Big|\; [x=z]P$$

where $x, y, z \in \mathcal{X}$ and $\vec{z}, \vec{y} \in \mathcal{X}^*$. The names $\vec{y}$ and y bind in the respective continuation process P. Here and below we suppose $\vec{y}$ contains no duplicated names. Process terms are taken up to alpha conversion. The typing rules, defining a judgement $\Gamma \vdash P$ to be read as 'P is typable with respect to Γ, structural congruence, written $\equiv$, and SOS are given in Figure 1. The SOS is similar to the 'early' definition of Sangiorgi [San93]. It differs in that it defines transitions of process terms equipped with a type context instead of partitioning the names into a subset for each type. This removes the need for side conditions on the free names of processes and simplifies the definitions of operational equivalences, as processes need only be compared with respect to the same type context. The *labels*, ranged over by α, are $\{\, \overline{x}\langle\vec{z}\rangle \mid x \in \mathcal{X} \wedge \vec{z} \in \mathcal{X}^* \,\} \cup \{\, x\langle\vec{z}\rangle \mid x \in \mathcal{X} \wedge \vec{z} \in \mathcal{X}^* \,\} \cup \{\tau\}$. The *names* of a label are $\mathrm{n}(\overline{x}\langle\vec{z}\rangle) = \{x\} \cup \vec{z}$, $\mathrm{n}(x\langle\vec{z}\rangle) = \{x\} \cup \vec{z}$ and $\mathrm{n}(\tau) = \{\}$. We define transition relations

$$\Gamma \vdash P \xrightarrow[\Delta]{\alpha} Q$$

where $\Gamma \vdash P$, the type context Δ contains names only at channel types, $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \{\}$, and Q is a process term. Intuitively Δ is the type context for the new names intruded or extruded by the transition (in the absence of subtyping Δ is determined by the other data). For example, if $\alpha = \overline{x}\langle z\rangle$ and $\Delta = \{z\!:\!\updownarrow\langle\rangle\}$ the transition above corresponds to a transition of [San93] with label $(\nu z)\overline{x}\langle z\rangle$, where z would be a name in the $\updownarrow\langle\rangle$ partition.

Proposition 1 *(Subject reduction) If* $\Gamma \vdash P \xrightarrow[\Delta]{\alpha} Q$ *then* $\Gamma, \Delta \vdash Q$*. Moreover,* $\mathrm{dom}(\Delta) \subseteq \mathrm{n}(\alpha)$*, if* $\alpha = \overline{x}\langle\vec{z}\rangle$ *then* $\Gamma(x) \leq {!}(\Gamma, \Delta)(\vec{z})$ *and if* $\alpha = x\langle\vec{z}\rangle$ *then* $\Gamma(x) \leq {?}(\Gamma, \Delta)(\vec{z})$*.*

Proposition 2 *(*CONG-L*) If* $P' \equiv P$ *and* $\Gamma \vdash P \xrightarrow[\Delta]{\alpha} Q$ *then* $\Gamma \vdash P' \xrightarrow[\Delta]{\alpha} Q$*.*

The current Pict libraries provide a rich set of primitives for interacting with Unix and the X window system. They are made available to programs by providing a pervasive type context Γ_{p} of certain channels along which a program can input and/or output. This includes, for example, a channel $\mathrm{print} : {!}\langle\mathrm{String}\rangle$ on which a program can output strings. The implementation will send these to standard output. *Programs* are process terms P such that $\Gamma_{\mathrm{p}} \vdash P$.

We now briefly describe the behaviour of the current Pict implementation. After type checking a Pict program is compiled to a C program which is then executed as a single Unix process. The implementation is thus sequential — the intended use of concurrency in Pict is for expressiveness, not for distributed or parallel programming (work on distribution is in progress). It maintains a *state* consisting of a *run queue* of processes to be scheduled (round robin) together with *channel queues* of processes waiting to communicate. It executes in steps, in each of which the process at the front of the run queue is removed and processed. This internal behaviour of the implementation is described by Turner in [Tur96, Ch. 7] and incorporated into the abstract machine given in §4. When an output or input on a library channel reaches the front of the run queue some special processing takes place. For many library channels this consists of a single call to a corresponding Unix IO routine. For example, processing $\overline{\mathrm{print}}\langle\text{“Ping”}\rangle$ involves an invocation of the C library call `printf(...)`. There are a number of facts that must be taken into account in order to accurately formalise a model of implementations and relate it to the SOS. We discuss them informally here, incorporating them into precise definitions in the following two sections.

Linear/Branching time The Pict SOS is nondeterministic. Any realistic implementation will be largely deterministic, however, as requiring that any nondeterministic path may be taken (stochastically, or as determined by an oracle) has a prohibitive performance cost for a programming language. Nonetheless, we do not wish to prescribe a particular scheduling strategy, as that would unduly prevent compiler optimisations. The SOS must therefore be regarded as a loose specification of the required implementation behaviour, so any definition of implementation correctness based on branching time, such as any notion of bisimulation, would render realistic implementations ‘incorrect’. Moreover, the strong forms of copying required to observe branching time distinctions [Mil81, Abr87] are not applicable to executing Pict implementations. One could, of course, examine an executing Pict implementation with a machine-level debugger. This would reveal many implementation-dependent details which the programmer and the language definition should abstract from, so we regard it as outside the intended use of the language.

Non-refusable communication All interaction between a Pict implementation and the operating system (which forms its immediate environment) occurs via invocations of C library calls, such as `printf(...)`, by the implementation. These calls cannot be ‘refused’ by the operating system and their return cannot be ‘refused’ by the

$$\textsc{Out}\ \frac{\Gamma(x) \leq !\langle \Gamma(z_1), \dots, \Gamma(z_n)\rangle}{\Gamma \vdash \overline{x}\langle z_1, \dots, z_n\rangle} \qquad \textsc{(Rep-)In}\ \frac{\begin{array}{c}\Gamma(x) \leq ?\langle T_1, \dots, T_n\rangle \\ \Gamma, y_1 : T_1, \dots, y_n : T_n \vdash P\end{array}}{\Gamma \vdash x(y_1, \dots, y_n).P} \\ \text{and} \quad \Gamma \vdash !\,x(y_1, \dots, y_n).P$$

$$\textsc{Par}\ \frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \,|\, Q} \qquad \textsc{Nil}\ \frac{}{\Gamma \vdash 0}$$

$$\textsc{Res}\ \frac{\begin{array}{c}\Gamma, x : T \vdash P \\ T \notin \mathcal{T}\end{array}}{\Gamma \vdash (\nu x : T)P} \qquad \textsc{Match}\ \frac{\begin{array}{c}\Gamma \vdash P \\ \Gamma(x) = \Gamma(y) \in \mathcal{T}\end{array}}{\Gamma \vdash [x = y]P}$$

$$\begin{array}{rcll}
P \,|\, 0 & \equiv & P & \\
P \,|\, Q & \equiv & Q \,|\, P & \\
P \,|\, (Q \,|\, R) & \equiv & (P \,|\, Q) \,|\, R & \\
(\nu x : T)(\nu y : T')P & \equiv & (\nu y : T')(\nu x : T)P & x \neq y \\
P \,|\, (\nu x : T)Q & \equiv & (\nu x : T)(P \,|\, Q) & x \notin \mathrm{fn}(P)
\end{array}$$

$$\textsc{Out}\ \frac{}{\Gamma \vdash \overline{x}\langle \vec{z}\rangle \xrightarrow[\{\}]{\overline{x}\langle \vec{z}\rangle} 0} \qquad \textsc{(Rep-)In}\ \frac{\begin{array}{c}\Gamma(x) \leq ?\,\vec{T} \\ (\Gamma, \Delta)(\vec{z}) = \vec{T} \\ \mathrm{dom}(\Delta) \subseteq \vec{z}\end{array}}{\Gamma \vdash x(\vec{y}).P \xrightarrow[\Delta]{x\langle \vec{z}\rangle} P[\vec{z}/\vec{y}]} \\ \text{and } \Gamma \vdash !\,x(\vec{y}).P \xrightarrow[\Delta]{x\langle \vec{z}\rangle} P[\vec{z}/\vec{y}] \,|\, !\,x(\vec{y}).P$$

$$\textsc{Par}\ \frac{\Gamma \vdash P \xrightarrow[\Delta]{\alpha} P'}{\Gamma \vdash P \,|\, Q \xrightarrow[\Delta]{\alpha} P' \,|\, Q} \qquad \textsc{Com}\ \frac{\Gamma \vdash P \xrightarrow[\Delta]{\overline{x}\langle \vec{z}\rangle} P' \quad \Gamma \vdash Q \xrightarrow[\Delta]{x\langle \vec{z}\rangle} Q'}{\Gamma \vdash P \,|\, Q \xrightarrow[\{\}]{\tau} (\nu \Delta)(P' \,|\, Q')}$$

$$\textsc{Res}\ \frac{\Gamma, x : T \vdash P \xrightarrow[\Delta]{\alpha} P' \quad x \notin \mathrm{n}(\alpha)}{\Gamma \vdash (\nu x : T)P \xrightarrow[\Delta]{\alpha} (\nu x : T)P'} \qquad \textsc{Open}\ \frac{\Gamma, x : T \vdash P \xrightarrow[\Delta]{\overline{w}\langle \vec{z}\rangle} P' \quad w \neq x \in \vec{z}}{\Gamma \vdash (\nu x : T)P \xrightarrow[\Delta, x : T]{\overline{w}\langle \vec{z}\rangle} P'}$$

$$\textsc{Match}\ \frac{\Gamma \vdash P \xrightarrow[\Delta]{\alpha} P'}{\Gamma \vdash [x = x]P \xrightarrow[\Delta]{\alpha} P'} \qquad \textsc{Cong-R}\ \frac{\Gamma \vdash P \xrightarrow[\Delta]{\alpha} P' \quad P' \equiv P''}{\Gamma \vdash P \xrightarrow[\Delta]{\alpha} P''}$$

Figure 1: Typing, structural congruence and structured operational semantics. Symmetrical versions of Par and Com are omitted.

implementation.

Blocking communication A Pict implementation should satisfy the following progress criterion: if a Pict program has at least one possible transition, i.e. a transition that is either an external input for which a value is available, an external output or an internal communication, then the implementation should perform one of the corresponding steps in a reasonable time. Now, some C library calls can potentially never return, e.g. `getchar`() if no characters become available. Such calls should therefore not be invoked unless either they can be guaranteed to return or the Pict program has no possible transitions. Consider, for example, the putative program $(\mathrm{getchar}(c).0 \mid \overline{\mathrm{print}}\langle\text{“Ping”}\rangle)$. An implementation should guarantee that the “Ping” is printed and so must not call `getchar`() first unless a character is available. (In fact we will consider only the most interesting potentially non-returning library call, for getting events from the X window system. Dealing with the others should not involve significant complication.)

External nondeterminism A C program cannot simultaneously ‘offer’ two library calls for the operating system to select between. An accurate model of implementations must therefore forbid external nondeterminism.

Termination When a Pict program terminates, i.e. when it has no more transitions in the SOS, the implementation Unix process terminates, typically returning the user to a Unix shell prompt. The user is therefore concerned with the termination of programs, despite the fact that there is no Pict language context that can ‘detect’ termination of an arbitrary subprogram.

Divergence Pict programs may diverge, i.e. have an infinite sequence of internal actions in the SOS. The user is concerned with the distinction between programs that diverge and programs that do not. Moreover, we cannot simply regard divergence as catastrophic, identifying divergent programs that are otherwise significantly different [Hoa85, Wal88]. To do so would allow an implementation to behave arbitrarily for any divergent program, which would be unduly confusing, particularly for programs which are unintentionally divergent. The testing scenario must therefore be divergence-sensitive.

Compositionality Much work on process calculi has been concerned with defining behavioural relations with good mathematical properties, e.g. congruence properties, axiomatisations, and coinductive characterisations. For this paper we take the problem of the development of tractable proof techniques to be secondary to that of giving a good language definition for Pict, although it must ultimately influence the language design. Now, Pict subprograms can only be composed (by parallel composition etc.) before they are compiled and only have behaviour when they are compiled and executing. There is therefore no reason to include congruence properties in the definition of correct program transformation, so it will be expressed simply in terms of a preorder over whole programs. For reasoning about particular programs one will obviously be concerned with *observational precongruence*, defined to be the largest precongruence contained in this preorder, and might ideally like a direct characterisation of it. We expect, however, that many program transformations are correct up to rather fine equivalences (see e.g. [PW96, NP96]) so for many purposes it will be preferable to have congruences with simple coinductive definitions that can be shown to be finer than observational precongruence.

X events Communication from the X window system to a C program, e.g. notification of mouse clicks, takes place via *X events*. These are generated and buffered within X. Two C library routines are provided; `XPending` returns the number of events in the buffer and `XNextEvent` returns the first available event. The latter may block, so access to it should not be provided directly to Pict programs (otherwise the progress criterion above will not be satisfiable). We therefore assume that access to X events is provided to Pict programs via a library channel $\mathrm{getXEvent}$. Inputs on this channel should not block the implementation unless there are no possible transitions.

Asynchrony Pict is based on an asynchronous π-calculus, in which outputs do not have continuation processes. This means that explicit acknowledgement signals must be used to control the sequencing of external IO. For example, instead of the program $\overline{\mathrm{print}}\langle\text{“Hello”}\rangle \mid \overline{\mathrm{print}}\langle\text{“World”}\rangle$, which could output “HelloWorld” or “WorldHello”, one can write $(\nu a : \updownarrow\langle\rangle)\ \big(\overline{\mathrm{pr}}\langle\text{“Hello”}, a\rangle \mid a().\overline{\mathrm{print}}\langle\text{“World”}\rangle\big)$. Here pr is a channel of type $!\langle\mathrm{String}, \updownarrow\langle\rangle\rangle$ provided by the libraries. An implementation executing the subprogram $\overline{\mathrm{pr}}\langle\text{“Hello”}, a\rangle$ must invoke the appropriate C library call and also add an acknowledgement $\overline{a}\langle\rangle$ to the program.

Fairness This paper leaves fairness properties for future work. Substantial effort has gone into ensuring that the current implementation is reasonably fair, as this is necessary for some natural programs. A good definition of implementation correctness should, therefore, require that the implementation is fair in some precise sense. One could also give a more accurate model of the interaction of a user and an implementation by taking the composition $\_ \,||\, \_$, defined in §3, to be a fair composition. In the distributed case the appropriate fairness properties will be more subtle, as will the issues of compositionality and external nondeterminism.

3 Implementation Model and Test Harnesses

In this section we give an abstract operational semantics of C programs, their environments, and the interactions between them. We define a class of models of the behaviour of C programs, ranged over by C, a test harness $\mathrm{H}_{\mathrm{am}}(\_)$ for these models, a class of models of the environments of C programs, ranged over by E, and a composition $\_ \,||\, \_$ of an environment and a C behaviour in the test harness. The behaviour of a composition $E \,||\, \mathrm{H}_{\mathrm{am}}(C)$ thus models the behaviour of an actual implementation. We then define a test harness $\mathrm{H}_{\mathrm{sos}}(\_)$ for Pict programs with the SOS semantics so that any $\mathrm{H}_{\mathrm{sos}}(P)$ and $\mathrm{H}_{\mathrm{am}}(C)$ are comparable. The definitions are given explicitly, rather than by encoding into some calculus, so that the facts from §2 can be incorporated directly.

We suppose that a C program and its environment can only interact in two ways. Firstly, the program can invoke an operating system (OS) routine, e.g. by executing a statement $\mathtt{y} = \mathtt{f}(\mathtt{a})$ for an OS routine `f` that takes an argument `a` and returns a result to be stored in `y`. Arguments and results may be values of some C structured types. We will assume that arguments and results are elements of the Pict base types, and so associate a pair of base types to each OS routine. This is obviously a major idealization for arbitrary C programs — we are not dealing with communication via shared memory or callback functions. For Pict implementations, however, it does not appear to be too serious. Secondly, the program can terminate by executing a statement `return`. A C program is single-threaded, so while an OS routine is executing the rest of the program is blocked. We can therefore model the program by a labelled transition system in which a single transition models a complete invocation of an OS routine.

We suppose, for each base type $t \in \mathcal{T}$, a non-empty set $|t|$ of its values. We assume that $\{\mathrm{Unit}, \mathrm{Nat}, \mathrm{XEvent}\} \subseteq \mathcal{T}$, $|\mathrm{Unit}| = \{\bullet\}$ and $|\mathrm{Nat}| = \mathbb{N}$. For technical simplicity we treat return as an OS routine of type $\langle \mathrm{Unit}, \mathrm{Unit}\rangle$. We use an 'early' LTS and do not distinguish input and output labels. We take a *C interface type context* Γ to be a finite partial function from $\mathcal{X}$ to pairs of base types. The *labels* over a C interface type context Γ are $\mathcal{L}(\Gamma) = \{\, x\langle a,r\rangle \mid \Gamma(x) = \langle A, R\rangle \wedge a \in |A| \wedge r \in |R| \,\} \cup \{\tau\}$. We take a labelled transition system S over a C interface type context Γ (a Γ-LTS) to consist of a set of states S, an initial state $\mathrm{root}(S) \in S$, and transitions $\xrightarrow{l} \subseteq S \times S$ for $l \in \mathcal{L}(\Gamma)$.

We will treat most OS routines uniformly, taking an arbitrary C interface type context Γ_u (strictly, with $\mathrm{dom}(\Gamma_\mathrm{u})$ not intersecting $\{\mathrm{return}, \mathrm{XPending}, \mathrm{XNextEvent}, \mathrm{XInsertEvent}\}$). X events and `return` must be treated specially, however. Letting Γ_c be $\Gamma_\mathrm{u}, \mathrm{return} : \langle \mathrm{Unit}, \mathrm{Unit}\rangle, \mathrm{XPending} : \langle \mathrm{Unit}, \mathrm{Nat}\rangle, \mathrm{XNextEvent} : \langle \mathrm{Unit}, \mathrm{XEvent}\rangle$, a C program behaviour will be modelled by a Γ_c-LTS. For $f : \langle A, R\rangle \in \Gamma_\mathrm{c}$, $a \in |A|$ and $r \in |R|$ a transition $c \xrightarrow{f\langle a,r\rangle} c'$ will model an invocation by the program in state c, of OS routine `f`, with argument a, returning result r, and with c' being the program state in which r has been returned.

Definition A *C-behaviour* C is a Γ_c-LTS satisfying conditions

1. If $f : \langle A, R\rangle \in \Gamma_\mathrm{c}$ and $c \xrightarrow{f\langle a,r\rangle} c_1$ then $\forall r' \in |R| \,.\, \exists c' \,.\, c \xrightarrow{f\langle a,r'\rangle} c'$.
2. If $\mathrm{root}(C) \xrightarrow{\mathcal{L}(\Gamma_\mathrm{c} - \mathrm{return})^*} \xrightarrow{\mathrm{return}\langle \bullet,\bullet\rangle} c'$ then $c' \not\longrightarrow$.
3. If $\mathrm{root}(C) \xrightarrow{\mathcal{L}(\Gamma_\mathrm{c} - \mathrm{return})^*} c$ then $c \longrightarrow$.
4. If $c \xrightarrow{l_1} c_1$ and $c \xrightarrow{l_2} c_2 \neq c_1$ then either $l_1 = l_2 = \tau$ or $\exists f, a, r_1, r_2 \,.\, l_1 = f\langle a, r_1\rangle \wedge l_2 = f\langle a, r_2\rangle \wedge r_1 \neq r_2$.

The conditions reflect the facts that a C program cannot 'refuse' a particular result value; that after an execution of `return` a C program has no behaviour; that a C program cannot 'halt' and that a C program cannot simultaneously 'offer' two OS routine calls for the OS to select between. It is arguable that one should also forbid internal nondeterminism (up to strong bisimilarity).

In order to compare a C-behaviour and the Pict SOS some semantic mismatches between them, involving X events, asynchrony and termination, must be addressed.

Simply relating transitions labelled $\mathrm{XNextEvent}\langle\bullet, ev\rangle$ of C-behaviours to transitions labelled $\mathrm{getXEvent}\langle ev\rangle$ of the Pict SOS would have two undesirable consequences. Firstly, it would allow an implementation to be 'correct' even though it could invoke `XNextEvent` with no events available and with other steps possible (hence not satisfying the progress criterion). Secondly, it would render some reasonable implementations, that get and internally buffer X events before the executing Pict program can input them, 'incorrect'. Instead, therefore, we will compare C-behaviours and the SOS in test harnesses that include explicit models of the X event buffer, allowing the OS to insert events at any time and a C-behaviour or SOS to remove events using their respective internal interfaces to the buffer. The test harness for C-behaviours is as follows, in which Γ_{e} is $\Gamma_{\mathrm{u}}, \mathrm{return}:\langle \mathrm{Unit}, \mathrm{Unit}\rangle, \mathrm{XInsertEvent}:\langle \mathrm{XEvent}, \mathrm{Unit}\rangle$, we write $|evs|$ for the length of the list evs and write $ev :: evs$ and $evs :: ev$ for the concatenation of the list evs with the singleton list ev.

Definition For a C-behaviour C we define $\mathrm{H}_{\mathrm{am}}(C)$ to be the Γ_{e}-LTS with states $\{\bullet\} \cup \{\, evs, c \mid evs \in |\mathrm{XEvent}|^* \wedge c \in C\,\}$, root $\langle \mathrm{nil}, \mathrm{root}(C)\rangle$ and transitions

$$\frac{c \xrightarrow{x\langle a,r\rangle} c' \quad x \in \mathrm{dom}(\Gamma_{\mathrm{u}})}{evs, c \xrightarrow{x\langle a,r\rangle} evs, c'} \qquad \frac{c \xrightarrow{\tau} c'}{evs, c \xrightarrow{\tau} evs, c'} \qquad \frac{c \xrightarrow{\mathrm{return}\langle\bullet,\bullet\rangle} c'}{evs, c \xrightarrow{\mathrm{return}\langle\bullet,\bullet\rangle} \bullet}$$

$$\frac{}{evs, c \xrightarrow{\mathrm{XInsertEvent}\langle ev,\bullet\rangle} (evs :: ev), c} \qquad \frac{c \xrightarrow{\mathrm{XPending}\langle\bullet,|evs|\rangle} c'}{evs, c \xrightarrow{\tau} evs, c'} \qquad \frac{c \xrightarrow{\mathrm{XNextEvent}\langle\bullet,ev\rangle} c'}{(ev :: evs), c \xrightarrow{\tau} evs, c'}$$

The environment of a C program, comprising Unix, the X window system (without its event buffer), a terminal etc., will also be modelled by an Γ_{e}-LTS. Its transitions model invocations by the program of non-X-event library routines, the program's final return and the insertion of events into the buffer by X.

Definition An *environment* E is a Γ_{e}-LTS satisfying

1. If $\mathrm{root}(E) \xrightarrow{\mathcal{L}(\Gamma_{\mathrm{e}} - \mathrm{return})^*} e$ then $\forall x :\langle A, R\rangle \in (\Gamma_{\mathrm{e}} - \mathrm{XInsertEvent})\ .\ \forall a \in |A|\ .\ \exists r \in |R|, e'\ .\ e \xrightarrow{x\langle a,r\rangle} e'$.
2. If $\mathrm{root}(E) \xrightarrow{\mathcal{L}(\Gamma_{\mathrm{e}} - \mathrm{return})^*} \xrightarrow{\mathrm{return}\langle\bullet,\bullet\rangle} e$ then $e \not\rightarrow$.

Condition 1 reflects the fact that Unix cannot 'refuse' a particular argument value of an OS routine call, or a `return`. Condition 2 reflects the fact that after an execution of `return` the environment has no further interaction with the program. The model of interactions between C programs and their environments is completed by a parallel composition operator:

Definition For an environment E and a Γ_{e}-LTS H we define $E \,||\, H$ to be the $\{\}$-LTS with states pairs e, h of states of E and H, root $\langle \mathrm{root}(E), \mathrm{root}(H)\rangle$ and transitions

$$\frac{e \xrightarrow{x\langle a,r\rangle} e' \quad h \xrightarrow{x\langle a,r\rangle} h'}{e, h \xrightarrow{\tau} e', h'} \qquad \frac{e \xrightarrow{\tau} e'}{e, h \xrightarrow{\tau} e', h} \qquad \frac{h \xrightarrow{\tau} h'}{e, h \xrightarrow{\tau} e, h'}$$

Turning to the SOS, the pervasive Pict type context Γ_{p} will be taken to have a name $x : !\langle A, \updownarrow\langle R\rangle\rangle$ for each $x :\langle A, R\rangle$ in Γ_{u}, together with $\mathrm{getXEvent} : ?\langle \mathrm{XEvent}\rangle$ and a name $x\!:\!t$ for each base type $t \in \mathcal{T}$ and $x \in |t|$. In example programs the types $!\langle A, \updownarrow\langle \mathrm{Unit}\rangle\rangle$ and $!\langle A, \updownarrow\langle\rangle\rangle$ will be confused. The test harness for the SOS must include a model of the X event buffer, must detect termination and must generate acknowledgements for asynchronous external IO.

Definition For a program P we define $\mathrm{H}_{\mathrm{sos}}(P)$ to be the Γ_{e}-LTS with states $\{\bullet\} \cup \{\, evs, P' \mid evs \in |\mathrm{XEvent}|^*$ and P' is a $\equiv$-class of programs$\,\}$, root $\langle \mathrm{nil}, P\rangle$ and transitions

$$\frac{\Gamma_{\mathrm{p}} \vdash P_1 \xrightarrow[y:\updownarrow\langle R\rangle]{\overline{x}\langle a,y\rangle} Q \qquad r \in |R|}{evs, P_1 \xrightarrow{x\langle a,r\rangle} evs, (\nu y : \updownarrow\langle R\rangle)(\overline{y}\langle r\rangle \mid Q)} \qquad \frac{\Gamma_{\mathrm{p}} \vdash P_1 \xrightarrow[\{\}]{\tau} P_2}{evs, P_1 \xrightarrow{\tau} evs, P_2} \qquad \frac{\Gamma_{\mathrm{p}} \vdash P_1 \not\longrightarrow}{evs, P_1 \xrightarrow{\mathrm{return}\langle\bullet,\bullet\rangle} \bullet}$$

$$\frac{}{evs, P_1 \xrightarrow{\mathrm{XInsertEvent}\langle ev,\bullet\rangle} (evs :: ev), P_1} \qquad \frac{\Gamma_{\mathrm{p}} \vdash P_1 \xrightarrow[\{\}]{\mathrm{getXEvent}\langle ev\rangle} P_2}{(ev :: evs), P_1 \xrightarrow{\tau} evs, P_2}$$

4 Testing

We would like to describe, to programmers and implementers, the behaviour of correct Pict implementations in terms of the SOS. The desired intuition is:

An implementation is correct if, for all programs P, a user interacting with the implementation running P cannot tell that he/she is not interacting with the LTS semantics of P given by the SOS.

To formalise 'user interacting with ... cannot tell ...' we consider the environments of §3 to be modelling the user of a Pict system, together with a terminal, Unix, and all of X except its event buffer. As the user is also the pertinent observer of the combined system, we can take observations based on changes of environment state. Letting H range over Γ_{e}-LTS's of the forms $\mathrm{H}_{\mathrm{am}}(C)$ and $\mathrm{H}_{\mathrm{sos}}(P)$, for C-behaviours C and programs P, and $\langle e, h\rangle$ range over states of $E \,||\, H$, we define:

$$\begin{aligned}
\langle e,h\rangle \rightharpoonup \langle e',h'\rangle &\overset{\mathit{def}}{\Leftrightarrow} e = e' \wedge \langle e,h\rangle \xrightarrow{\tau} \langle e',h'\rangle \\
\langle e,h\rangle \rightharpoondown \langle e',h'\rangle &\overset{\mathit{def}}{\Leftrightarrow} e \neq e' \wedge \langle e,h\rangle \xrightarrow{\tau} \langle e',h'\rangle \\
\langle e,h\rangle \mathrel{=\!\!\!\rightharpoondown} \langle e',h'\rangle &\overset{\mathit{def}}{\Leftrightarrow} \langle e,h\rangle \rightharpoonup^* \rightharpoondown \rightharpoonup^* \langle e',h'\rangle
\end{aligned}$$

As discussed in §2 the definitions of correctness should be linear time and sensitive to termination and divergence. We therefore take *tests o* to be $\langle E, \vec{e}, \mathrm{par}\rangle$, $\langle E, \vec{e}, \mathrm{term}\rangle$, $\langle E, \vec{e}, \mathrm{div}\rangle$ and $\langle E, \tilde{e}\rangle$, where E is an environment, $\vec{e} \in E^*$ and $\tilde{e} \in E^\omega$. We define a may-testing preorder $\sqsubseteq$ as follows.

$$\begin{aligned}
H \;\mathbf{may}\; \langle E, \vec{e}, \mathrm{par}\rangle &\overset{\mathit{def}}{\Leftrightarrow} \exists \vec{h} \,.\, \langle \mathrm{root}(E), \mathrm{root}(H)\rangle \mathrel{=\!\!\!\rightharpoondown} \langle e_1, h_1\rangle \ldots \mathrel{=\!\!\!\rightharpoondown} \langle e_n, h_n\rangle \\
H \;\mathbf{may}\; \langle E, \vec{e}, \mathrm{term}\rangle &\overset{\mathit{def}}{\Leftrightarrow} \exists \vec{h} \,.\, \langle \mathrm{root}(E), \mathrm{root}(H)\rangle \mathrel{=\!\!\!\rightharpoondown} \ldots \mathrel{=\!\!\!\rightharpoondown} \langle e_n, h_n\rangle \rightharpoonup^* \not\longrightarrow \\
H \;\mathbf{may}\; \langle E, \vec{e}, \mathrm{div}\rangle &\overset{\mathit{def}}{\Leftrightarrow} \exists \vec{h} \,.\, \langle \mathrm{root}(E), \mathrm{root}(H)\rangle \mathrel{=\!\!\!\rightharpoondown} \ldots \mathrel{=\!\!\!\rightharpoondown} \langle e_n, h_n\rangle \rightharpoonup^\omega \\
H \;\mathbf{may}\; \langle E, \tilde{e}\rangle &\overset{\mathit{def}}{\Leftrightarrow} \exists \tilde{h} \,.\, \langle \mathrm{root}(E), \mathrm{root}(H)\rangle \mathrel{=\!\!\!\rightharpoondown} \langle e_1, h_1\rangle \mathrel{=\!\!\!\rightharpoondown} \langle e_2, h_2\rangle \ldots \\
H \sqsubseteq H' &\overset{\mathit{def}}{\Leftrightarrow} \forall o \,.\, H \;\mathbf{may}\; o \Rightarrow H' \;\mathbf{may}\; o
\end{aligned}$$

Finally we can give our central definition:

Definition An *abstract machine* M is a function from programs to C-behaviours. It is *correct* if for all programs P we have $\mathrm{H}_{\mathrm{am}}(M(P)) \sqsubseteq \mathrm{H}_{\mathrm{sos}}(P)$.

We now briefly discuss $\sqsubseteq$ and some conceivable alternatives. It is perhaps the finest appropriate notion. There are two obvious ways in which its definition could be weakened, by amalgamating the termination and divergence tests and by omitting the infinite tests. We have chosen not to do either, as we expect that any reasonable abstract machine will be correct up to the stronger notion given (this is supported by Theorem 1 below). We therefore might as well provide programmers with the stronger guarantees about behaviour. The preorder is not affected by the omission of the tests $\langle E, \vec{e}, \mathrm{par}\rangle$. It can be given a direct characterization as an annotated trace inclusion. For $k \in \mathcal{L}(\Gamma_{\mathrm{e}}) - \{\tau\}$ we take $\xRightarrow{k} \overset{\mathit{def}}{=} \xrightarrow{\tau}^{*} \xrightarrow{k} \xrightarrow{\tau}^{*}$ as usual.

$$\begin{array}{rcl}
\mathrm{tr}_{\mathrm{dead}}(H) & \overset{\mathit{def}}{=} & \{\, k_1 \ldots k_n \mid \exists h \,.\, \mathrm{root}(H) \xRightarrow{k_1} \ldots \xRightarrow{k_n} \xrightarrow{\tau}^{*} h \\
& & \qquad \wedge \neg \exists l \in \mathcal{L}(\Gamma_{\mathrm{e}} - \mathrm{XInsertEvent}) \,.\, h \xrightarrow{l} \} \\
\mathrm{tr}_{\mathrm{div}}(H) & \overset{\mathit{def}}{=} & \{\, k_1 \ldots k_n \mid \mathrm{root}(H) \xRightarrow{k_1} \ldots \xRightarrow{k_n} \xrightarrow{\tau}^{\omega} \} \\
\mathrm{tr}_{\mathrm{inf}}(H) & \overset{\mathit{def}}{=} & \{\, k_1 k_2 \ldots \mid \mathrm{root}(H) \xRightarrow{k_1} \xRightarrow{k_2} \ldots \}
\end{array}$$

Proposition 3 $H \sqsubseteq H'$ *iff* $\forall \kappa \in \{\mathrm{dead}, \mathrm{div}, \mathrm{inf}\} \,.\, \mathrm{tr}_\kappa(H) \subseteq \mathrm{tr}_\kappa(H')$.

The proof of this uses standard techniques, involving discriminating environments constructed from traces over $\mathcal{L}(\Gamma_{\mathrm{e}}) - \{\tau\}$. The preorder $\sqsubseteq$ is intuitively closely related to the annotated trace inclusions of Hoare [Hoa85] and the testing of De Nicola and Hennessy [DH84]. We give an exact comparison with the latter. Instantiating their definitions, say a *DH-test* d is a pair $\langle E, \surd\rangle$ of an environment E and set of 'successful' states $\surd \subseteq E$. Letting $e_0 = \mathrm{root}(E)$:

$$\begin{array}{rcl}
H \ \mathbf{may}\ \langle E, \surd\rangle & \overset{\mathit{def}}{\Leftrightarrow} & \exists e \in \surd, h \,.\, \langle e_0, \mathrm{root}(H)\rangle \longrightarrow^{*} \langle e, h\rangle \\
H \ \mathbf{must}\ \langle E, \surd\rangle & \overset{\mathit{def}}{\Leftrightarrow} & \langle e_0, \mathrm{root}(H)\rangle \xrightarrow{\tau} \ldots \xrightarrow{\tau} \langle e_n, h_n\rangle \not\longrightarrow \Rightarrow \exists i \in 0..n \,.\, e_i \surd \\
& & \wedge\ \langle e_0, \mathrm{root}(H)\rangle \xrightarrow{\tau} \langle e_1, h_1\rangle \xrightarrow{\tau} \ldots \Rightarrow \exists i \geq 0 \,.\, e_i \surd
\end{array}$$

$$\begin{array}{rcl}
H \sqsubseteq_{\mathrm{DH-may}} H' & \overset{\mathit{def}}{\Leftrightarrow} & \forall d \,.\, H\ \mathbf{may}\ d \Rightarrow H'\ \mathbf{may}\ d \\
H \sqsubseteq_{\mathrm{DH-must}} H' & \overset{\mathit{def}}{\Leftrightarrow} & \forall d \,.\, H\ \mathbf{must}\ d \Rightarrow H'\ \mathbf{must}\ d
\end{array}$$

It is straightforward to check that $\sqsubseteq \subsetneq (\sqsubseteq_{\mathrm{DH-may}} \cap \sqsubseteq_{\mathrm{DH-must}}^{-1})$. For forwards must-testing, however, one has $\sqsubseteq \not\subseteq \sqsubseteq_{\mathrm{DH-must}}$. Moreover, including forwards must-testing would render reasonable abstract machines 'incorrect', e.g. abstract machines that, given the program $(\nu x : \updownarrow\langle\rangle)(\overline{x}\langle\rangle \mid x().0 \mid x().(\nu y : \updownarrow\langle\rangle)(\overline{y}\langle\rangle \mid {!}\,y().\overline{y}\langle\rangle))$, always return.

One could imagine taking tests based on Pict contexts (following e.g. [Hen91, BD95]) rather than on our test harnesses and $||$. It is hard to see how the facts about Pict could be introduced, particularly given that termination detection is required. More generally, there is no reason to suppose that the interactions between a Pict implementation and its environment are of the same kind as the interactions between two Pict subprograms. Several authors have considered restricting to observations that are in some sense effective. A user may indeed only be concerned with effective *real time* properties of implementations, however we do not wish to incorporate these properties into the language definition. We must therefore either allow 'infinite' observations or neglect divergence despite the fact that users are concerned with it.

We give an example abstract machine M_1, closely based on the actual Pict implementation, in Figure 2. It has states $\langle \Gamma \,;\, rq \,;\, cqs \rangle$ consisting of a type context Γ, a run queue rq and a collection of channel queues cqs. The rules for internal transitions follow those of Turner [Tur96, Ch. 7], although without certain optimisations. For external communication X is polled periodically for new X events (whenever the token $\mathbb{X}$ reaches the front of the run queue). The rules ensure that no potentially blocking $\mathrm{XNextEvent}$ call is performed except when all processes are waiting for an event, in which case a blocking call *is* made (to avoid busy-waiting). Other non-blocking IO occurs immediately, and the machine returns when there are no processes left in the run queue or left waiting for X events.

Theorem 1 *M_1 is a correct abstract machine.*

The proof of this uses a decompilation function from states of $\mathrm{H}_{\mathrm{am}}(M_1(P))$ to states of $\mathrm{H}_{\mathrm{sos}}(P)$, using which one can prove that $\mathrm{H}_{\mathrm{am}}(M_1(P))$ simulates (in a specialised sense that refines $\sqsubseteq$) $\mathrm{H}_{\mathrm{sos}}(P)$. The treatment of events and divergence is delicate; the rest is reasonably straightforward (as one would hope, as M_1 is not optimised).

5 The Observational Precongruence

The same testing scenario can be used to define an observational preorder over programs that is appropriate as a standard of correctness for program transformations, such as those considered by Jones [Jon96], Philippou and Walker [PW96], and Nestmann and Pierce [NP96]. The desired intuition is:

A program P can be transformed to a program P' if, for all correct abstract machines M, a user interacting with $\mathrm{H}_{\mathrm{am}}(M(P'))$ cannot tell that he/she is not interacting with $\mathrm{H}_{\mathrm{sos}}(P)$.

This is formalised by the *observational preorder* $\dot{\gtrsim}$ over programs, where $P \dot{\gtrsim} P'$ iff for all correct abstract machines M we have $\mathrm{H}_{\mathrm{am}}(M(P')) \sqsubseteq \mathrm{H}_{\mathrm{sos}}(P)$. (Transitivity requires a brief argument.) The observational preorder reflects the realities of Pict implementations, with for example $(\nu x : \updownarrow\langle\rangle)\overline{\mathrm{pr}}\langle\text{“Ping”}, x\rangle \;\; (\dot{\gtrsim} \cap \dot{\gtrsim}^{-1})$ $(\nu x : \updownarrow\langle\rangle)(\overline{\mathrm{pr}}\langle\text{“Ping”}, x\rangle \,|\, x().x().P)$, $(\nu x : \updownarrow\langle\rangle)(\overline{x}\langle\rangle \,|\, x().0 \,|\, x().\overline{\mathrm{pr}}\langle\text{“Ping”}, x\rangle) \;\; \dot{\gtrsim} \;\; 0$ and $(\nu x : \updownarrow\langle\rangle)\overline{\mathrm{pr}}\langle\text{“Ping”}, x\rangle \dot{\not\gtrsim} 0$. These confirm that $\dot{\gtrsim}$ differs from standard π-calculus notions. As correctness does not fix a particular scheduling strategy, a correct abstract machine may have completely different behaviour when executing structurally congruent programs, let alone those related by $\dot{\gtrsim}$. Further, different correct abstract machines may behave completely differently when executing the same program. A programmer should therefore only write programs for which any execution path (from

STATES: $\langle \Gamma\,;\, rq\,;\, cqs\rangle$ where

- Γ is a finite type context with names only at channel types such that $\mathrm{dom}(\Gamma_{\mathrm{p}}) \cap \mathrm{dom}(\Gamma) = \{\}$
- $\Gamma'_{\mathrm{p}} = (\Gamma_{\mathrm{p}} - \mathrm{getXEvent}), \mathrm{getXEvent} : \updownarrow\langle \mathrm{XEvent}\rangle$
- $\mathrm{outs}(x,\Gamma) = \{\, \overline{x}\langle\vec{z}\rangle \mid \Gamma'_{\mathrm{p}}, \Gamma \vdash \overline{x}\langle\vec{z}\rangle \,\}^{+}$
- $\mathrm{ins}(x,\Gamma) = (\{\, x(\vec{y}).P \mid \Gamma'_{\mathrm{p}}, \Gamma \vdash x(\vec{y}).P \,\} \cup \{\, !\,x(\vec{y}).P \mid \Gamma'_{\mathrm{p}}, \Gamma \vdash !\,x(\vec{y}).P \,\})^{+}$
- $rq \in (\{\mathbb{X}, \mathbb{X}'\} \cup \{\, P \mid \Gamma'_{\mathrm{p}}, \Gamma \vdash P \,\})^{*}$
- $cqs \in \Pi x \in \mathrm{dom}(\Gamma) \cup \{\mathrm{getXEvent}\}\ .\ \{\mathrm{nil}\} \cup \mathrm{outs}(x,\Gamma) \cup \mathrm{ins}(x,\Gamma)$

INITIAL STATE: $\mathrm{root}(M_1(P_0)) \overset{def}{=} \langle \{\}\,;\, P_0 :: \mathbb{X} :: \mathrm{nil}\,;\, \{\mathrm{getXEvent} \mapsto \mathrm{nil}\}\rangle.$

TRANSITIONS:

$$
\begin{array}{lll}
\langle \Gamma\,;\, 0 :: rq\,;\, cqs\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq\,;\, cqs\rangle & \\
\langle \Gamma\,;\, \overline{x}\langle\vec{z}\rangle :: rq\,;\, cqs, x \mapsto x(\vec{y}).P :: cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq :: P[\vec{z}/\vec{y}]\,;\, cqs, x \mapsto cq\rangle & \\
\langle \Gamma\,;\, \overline{x}\langle\vec{z}\rangle :: rq\,;\, cqs, x \mapsto !\,x(\vec{y}).P :: cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq :: P[\vec{z}/\vec{y}] :: !\,x(\vec{y}).P\,;\, cqs, x \mapsto cq\rangle & \\
\langle \Gamma\,;\, \overline{x}\langle\vec{z}\rangle :: rq\,;\, cqs, x \mapsto cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq\,;\, cqs, x \mapsto cq :: \overline{x}\langle\vec{z}\rangle\rangle & 1 \\
\langle \Gamma\,;\, x(\vec{y}).P :: rq\,;\, cqs, x \mapsto \overline{x}\langle\vec{z}\rangle.P :: cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq :: P[\vec{z}/\vec{y}]\,;\, cqs, x \mapsto cq\rangle & \\
\langle \Gamma\,;\, x(\vec{y}).P :: rq\,;\, cqs, x \mapsto cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq\,;\, cqs, x \mapsto cq :: x(\vec{y}).P\rangle & 2 \\
\langle \Gamma\,;\, !\,x(\vec{y}).P :: rq\,;\, cqs, x \mapsto \overline{x}\langle\vec{z}\rangle.P :: cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq :: P[\vec{z}/\vec{y}] :: !\,x(\vec{y}).P\,;\, cqs, x \mapsto cq\rangle & \\
\langle \Gamma\,;\, !\,x(\vec{y}).P :: rq\,;\, cqs, x \mapsto cq\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq\,;\, cqs, x \mapsto cq :: !\,x(\vec{y}).P\rangle & 2 \\
\langle \Gamma\,;\, (P \mid Q) :: rq\,;\, cqs\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq :: P :: Q\,;\, cqs\rangle & \\
\langle \Gamma\,;\, (\nu x : T)P :: rq\,;\, cqs\rangle & \xrightarrow{\tau} \langle (\Gamma, x : T)\,;\, rq :: P\,;\, cqs, x \mapsto \mathrm{nil}\rangle & 3 \\
\langle \Gamma\,;\, [x = x]P :: rq\,;\, cqs\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq :: P\,;\, cqs\rangle & \\
\langle \Gamma\,;\, [x = y]P :: rq\,;\, cqs\rangle & \xrightarrow{\tau} \langle \Gamma\,;\, rq\,;\, cqs\rangle & 4 \\
 & & \\
\langle \Gamma\,;\, \overline{x}\langle a, y\rangle :: rq\,;\, cqs\rangle & \xrightarrow{x\langle a, r\rangle} \langle \Gamma\,;\, rq :: \overline{y}\langle r\rangle\,;\, cqs\rangle & 5 \\
 & & \\
\langle \Gamma\,;\, \mathbb{X} :: P :: rq\,;\, cqs\rangle & \xrightarrow{\mathrm{XPending}\langle \bullet, 0\rangle} \langle \Gamma\,;\, P :: rq :: \mathbb{X}\,;\, cqs\rangle & \\
\langle \Gamma\,;\, \mathbb{X} :: P :: rq\,;\, cqs\rangle & \xrightarrow{\mathrm{XPending}\langle \bullet, n+1\rangle} \langle \Gamma\,;\, \mathbb{X}' :: P :: rq\,;\, cqs\rangle & \\
\langle \Gamma\,;\, \mathbb{X}' :: P :: rq\,;\, cqs\rangle & \xrightarrow{\mathrm{XNextEvent}\langle \bullet, ev\rangle} \langle \Gamma\,;\, \overline{\mathrm{getXEvent}}\langle ev\rangle :: P :: rq :: \mathbb{X}\,;\, cqs\rangle & \\
\langle \Gamma\,;\, \mathbb{X} :: \mathrm{nil}\,;\, cqs\rangle & \xrightarrow{\mathrm{XNextEvent}\langle \bullet, ev\rangle} \langle \Gamma\,;\, \overline{\mathrm{getXEvent}}\langle ev\rangle :: \mathbb{X} :: \mathrm{nil}\,;\, cqs\rangle & 6 \\
\langle \Gamma\,;\, \mathbb{X} :: \mathrm{nil}\,;\, cqs\rangle & \xrightarrow{\mathrm{return}\langle \bullet, \bullet\rangle} \langle \Gamma\,;\, \mathrm{nil}\,;\, cqs\rangle & 7
\end{array}
$$

Side conditions 1: $cq \notin \mathrm{ins}(x,\Gamma)$, 2: $cq \notin \mathrm{outs}(x,\Gamma)$, 3: $x \notin \mathrm{dom}(\Gamma_{\mathrm{p}}, \Gamma)$, 4: $x \neq y$,
5: $\Gamma_{\mathrm{p}}(x) = !\langle A, \updownarrow\langle R\rangle\rangle \wedge r \in |R|$, 6: $cqs(\mathrm{getXEvent}) \in \mathrm{ins}(\mathrm{getXEvent},\Gamma)$,
7: $cqs(\mathrm{getXEvent}) \notin \mathrm{ins}(\mathrm{getXEvent},\Gamma)$.

Figure 2: The abstract machine M_1 executing a program P_0

those given by the SOS) would be acceptable. This is obviously problematic, accentuating the need for formal development techniques and the identification of well behaved (especially, confluent) idioms.

The observational preorder induces an *observational precongruence* $\gtrsim$ and hence a term model for Pict. Say a *typed precongruence* $>$ is a family of relations, indexed by type contexts, such that each $>_\Gamma$ is an preorder over $\{\, P \mid \Gamma \vdash P \,\}$ and $>$ satisfies the evident congruence rules. $\gtrsim$ is defined to be the largest typed precongruence such that $\gtrsim_{\Gamma_\mathrm{p}} \subseteq \dot{\gtrsim}$. We conclude by giving a typed congruence with a tractable definition that refines $\gtrsim$ and may suffice for many program transformations. Further work should take into account the asynchronous nature of Pict and a larger fragment of its type system, building on [ACS96, PS96]. Say *weak divergence bisimulation*, written $\dot{\approx}$, is the largest type context indexed family of relations such that each $\dot{\approx}_\Gamma$ is a symmetric relation over $\{\, P \mid \Gamma \vdash P \,\}$ and, for all $P \dot{\approx}_\Gamma Q$: if $\Gamma \vdash P \xrightarrow[\Delta]{\alpha} P'$ then $\exists Q' \,.\, \Gamma \vdash Q \overset{\hat{\alpha}}{\underset{\Delta}{\Longrightarrow}} Q' \wedge P' \dot{\approx}_{\Gamma,\Delta} Q'$ and if $\Gamma \vdash P \xrightarrow[\{\}]{\tau}{}^{\omega}$ then $\Gamma \vdash Q \xrightarrow[\{\}]{\tau}{}^{\omega}$, where $\overset{\hat{\alpha}}{\underset{\Delta}{\Longrightarrow}} \overset{def}{=} \xrightarrow[\{\}]{\tau}{}^{*} \xrightarrow[\Delta]{\alpha} \xrightarrow[\{\}]{\tau}{}^{*}$, for $\alpha \neq \tau$, and $\overset{\hat{\tau}}{\underset{\{\}}{\Longrightarrow}} \overset{def}{=} \xrightarrow[\{\}]{\tau}{}^{*}$. We define $\approx$ by $P \approx_\Gamma Q$ iff for all substitutions $[\vec{z}/\vec{y}]$, such that $\forall i \,.\, \Gamma(z_i) = \Gamma(y_i)$, we have $P[\vec{z}/\vec{y}] \dot{\approx}_\Gamma Q[\vec{z}/\vec{y}]$.

Proposition 4 $\approx$ *is a typed congruence.*

Theorem 2 $\approx \subseteq \gtrsim$.

Acknowledgements I would like to thank Benjamin Pierce for many interesting discussions on this work and the EPSRC for support via grant GR/K 38403.

References

[Abr87] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[ACS96] Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. In Montanari and Sassone [MS96], pages 147–162.

[AV93] S. Abramsky and S. J. Vickers. Quantales, observational logic and process semantics. *Mathematical Structures in Computer Science*, 3:161–227, 1993.

[BD95] Michele Boreale and Rocco De Nicola. Testing equivalences for mobile processes. *Information and Computation*, 120:279–303, 1995.

[DH84] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[FG96] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In POPL'96 [POP96], pages 372–385.

[Gla90] R. J. van Glabeek. The linear time — branching time spectrum. In *Proceedings CONCUR '90, LNCS 458,* pages 278–297, 1990.

[Gla93] R. J. van Glabbeek. The linear time — branching time spectrum II (the semantics of sequential systems with silent moves). In *Proceedings of CONCUR '93, LNCS 715*, pages 66–81, 1993.

[Hen91] M. Hennessy. A model for the π-calculus. Technical Report 91:08, University of Sussex, 1991.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *Proceedings 7th ICALP, LNCS 85*, pages 299–309, 1980.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, 1985.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In POPL'96 [POP96], pages 295–308.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Mil81] Robin Milner. A modal characterisation of observable machine-behaviour. In *Proceedings CAAP '81, LNCS 112*, pages 25–34, 1981.

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

[MS96] Ugo Montanari and Vladimiro Sassone, editors. *Proceedings CONCUR 96,* Pisa, Italy, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[NP96] Uwe Nestmann and Benjamin C. Pierce. Decoding choice encodings. In Montanari and Sassone [MS96], pages 179–194.

[POP96] *Conference Record of the 23^{rd} ACM Symposium on Principles of Programming Languages*, 1996.

[PS96] Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.

[PT97] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. Technical report, Computer Science Department, Indiana University, 1997. To appear in Milner *festschrift*, MIT Press.

[PW96] Anna Philippou and David Walker. On transformations of concurrent object programs. In Montanari and Sassone [MS96], pages 131–146.

[Rep92] John Hamilton Reppy. *Higher-Order Concurrency*. PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.

[San93] Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1993.

[TLK96] Bent Thomsen, Lone Leth, and Tsung-Min Kuo. A Facile tutorial. In Montanari and Sassone [MS96], pages 278–298.

[Tur96] David N. Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1996.

[Wal88] D. J. Walker. Bisimulations and divergence. In *Proc. 3rd IEEE Symposium on Logic in Computer Science*, pages 186–192, 1988.