About the Completeness of Type Systems

Stefan Kahrs*

University of Edinburgh
Laboratory for Foundations of Computer Science
King's Buildings,
Edinburgh EH9 3JZ
United Kingdom
email: smk@dcs.ed.ac.uk

Abstract

The original purpose of type systems for programming languages was to prevent certain forms of run-time errors, like using a number as a function. Some type systems go as far as guaranteeing the absence of run-time errors, e.g. the type system of Standard ML. One can call such a type system "sound".

This raises the question of the dual notion of completeness, i.e. is everything typable that does not have run-time errors? Or, to put it in another way: does the type system restrict the expressive power of the underlying implementation in an undesirable way?

To make this rather vague idea precise we define an abstract notion of "type system", together with general notions of soundness and completeness. We examine several type systems for these properties, for instance λ^τ and PCF are both complete, but for very different reasons.

1 Introduction

The purpose of type systems for programming languages was originally to prevent run-time errors like using a character as a pointer, etc. Nowadays people have found additional uses of types, especially following the propositions-as-types principle. However, for this paper our main interest in type systems is the way they prevent errors, not in what way the typable terms can be viewed as proof objects.

Robin Milner once stated the slogan "well-typed programs don't go wrong". This actually means that (if the type system is sound then) programs that type-check do not have run-time errors. Without the parentheses the statement is a tautology, as this is just the definition of soundness. Thus, Milner's statement

*The research reported here was supported by SERC grant GR/J07303.

asserts the soundness of some type system, here it probably was the type system of ML.

The above paragraph rests on one unexplained notion, the notion of run-time error. What actually *is* a run-time error? This is not and cannot be an absolute notion; take for example the familiar "core dumped" error: if a program terminates with message "core dumped" then it is unclear whether we had a *real* run-time error or whether our program simply printed "core dumped" on the screen and terminated. We need to define which run-time scenarios we regard as run-time errors. This is to some extent a matter of taste, we have a design decision to make. Having said that, in most cases there is an obvious choice.

Trivially, an operational semantics without a notion of run-time error has only sound type systems. But this is not very helpful to the programmer who does make errors from time to time; not all these errors can be detected by the type system, but quite a few (the majority, actually) can if the type system is rich enough.

Suppose we have a run-time system plus a sound type system. Can we turn Milner's slogan around, is it true that "programs that don't go wrong are well-typed?" This is the completeness question for the type system which is the subject of this paper. The question matters as the completeness of a type system means that the expressive power of the underlying operational semantics is not restricted by the type system.

To be a bit more precise: we are talking about unrestricted expressive power *w.r.t. the types the type system provides.* In general this is quite different from the expressive power of the untyped language itself. For example, the simply typed λ-calculus λ^τ is not Turing-complete, while the untyped λ-calculus is, but this does not directly imply the incompleteness of the simple type system. The problem is that the missing terms would have to be given simple types, but it is not obvious that we could do so without introducing errors (whatever we have declared to be an error).

Moreover, when we assess the expressive power of a language it is not so much our concern which *program texts* pass the type-checker as which *programs* do, where a program is simply an equivalence class of observationally equivalent program descriptions. We shall call a type system *bicomplete* whenever all sound programs are expressible up to observational equivalence; for *completeness*, we do not require equivalence but rather some information preorder between the programs. Intuition: think of the partial order in domain theory — our notion of completeness is satisfied if we can always express a "larger" program.

2 Typing Transition Systems

In order to talk about arbitrary type systems for arbitrary programming languages in any non-informal way we need an abstract notion of programming language and an abstract notion of type system.

We shall first focus on the untyped world. To avoid too specific notions of abstract programming language we firmly base it on the notion of labelled transition system, see e.g. [16, 11, 5].

Definition 1
A *transition system* is a structure (Sta, Lab, Tra) where

- Sta is a set of *states*,
- Lab is a set of *labels*, and
- $Tra \subset Sta \times Lab \times Sta$ is the *transition relation*.

A transition system is called *pointed* if there is a distinguished initial state ι.

As usual, we write $s \xrightarrow{l} s'$ for $(s, l, s') \in Tra$; also, $s \xrightarrow{w} s'$ with $w \in Lab^*$ is shorthand for that there exist states $s_0, \ldots, s_n$ and labels $l_1, \ldots, l_n$ with $s = s_0$, $s' = s_n$, $w = l_1 \cdots l_n$ and $s_{i-1} \xrightarrow{l_i} s_i$. If $s \xrightarrow{w} s'$ then s' is *reachable from* s; a state is *reachable* iff it is reachable from the initial state ι. We write $\twoheadrightarrow$ for the reachability relation, i.e. $s \twoheadrightarrow s' \iff \exists w \in Lab^*.\, s \xrightarrow{w} s'$.

Transition systems are traditionally studied as models for concurrent programming, nondeterministic programming, etc. This is not our main objective here, in particular we are interested in the ordinary deterministic scenario:

Definition 2
An TS is called *deterministic* if $s \xrightarrow{l} s'$ and $s \xrightarrow{l} s''$ imply $s' = s''$.

Definition 3
Given a transition system, a relation $\lesssim$ on states is a *simulation* if the following holds: If $s_1 \lesssim s_2$ then

$$\forall l \in Lab, s_1' \in Sta.\, (s_1 \xrightarrow{l} s_1' \;\supset\; \exists s_2' \in Sta.\, (s_2 \xrightarrow{l} s_2' \wedge s_1' \lesssim s_2')).$$

A *bisimulation* is a simulation $\lesssim$ such that $\gtrsim$ is a simulation as well.

The notions of simulation and bisimulation are usually studied in the context of process algebra [5] or modal logic [11]. However, the use of bisimulations we have in mind in this paper is better indicated by Rutten's observation [14] of the duality between algebras and congruence relations on one hand and transition systems (viewed as coalgebras) and bisimulations on the other.

Here are a few useful observations on simulations and bisimulations: simulations are closed under (arbitrary) union and composition, hence the transitive closure of a simulation is a simulation and there is always a largest simulation. We write $\precapprox$ for the largest simulation in a TS (which is necessarily a preorder) and $\approx$ for the largest bisimulation (which is necessarily an equivalence). The reflexive transitive closure of a simulation is also a simulation, since the identity relation obviously is one. Bisimulations are also closed under inversion and hence under equivalence closure or partial equivalence closure. The symmetric interior of a simulation is a bisimulation whenever the TS is deterministic.

There are several ways of defining an "information ordering" for any transition system, one is the familiar order on traces:

Definition 4
We define a relation $\sqsubseteq$ on states as follows:

$$s_1 \sqsubseteq s_2 \iff \forall w \in \mathit{Lab}^*, s_1' \in \mathit{Sta}.\, s_1 \xrightarrow{w} s_1' \supset \exists s_2' \in \mathit{Sta}.\, s_2 \xrightarrow{w} s_2'$$

We write $s \equiv s'$ for $s \sqsubseteq s' \wedge s' \sqsubseteq s$.

Proposition 1 *The relation $\sqsubseteq$ is a preorder.*

Simulations and the trace preorder are connected, especially for deterministic or even *pseudo-deterministic* transition systems. We call a TS pseudo-deterministic iff $s \xrightarrow{l} s'$ and $s \xrightarrow{l} s''$ always imply $s' \equiv s''$.

Proposition 2 *Any simulation is contained in $\sqsubseteq$ and any bisimulation in $\equiv$. If the TS is pseudo-deterministic then $\sqsubseteq = \precapprox$ and $\equiv = \approx$.*

We take pointed transition systems as our abstract notion of programming language, with one modification: we introduce a notion of error-state in order to be able to express soundness.

Definition 5
An error transition system is a pair (T, Err) such that T is a transition system $(\mathit{Sta}, \mathit{Lab}, \mathit{Tra})$ and $\mathit{Err} \subseteq \mathit{Sta}$ is a set of *error states*. The *strict part* of an ETS is the TS T_0 obtained from T by removing all error states and restricting the transition relation accordingly.

The strict part of a pointed error transition system is pointed if ι is not an error state itself. A simulation (bisimulation) of an ETS is a simulation (bismulation) of its strict part. For error transition systems there is a simple notion of soundness:

Definition 6
A state s is *sound* iff no error state is reachable from s. A pointed ETS is sound if no error state is reachable. For each ETS $T = (X, \mathit{Err})$ we define another ETS $T^* = (X, \{s \in \mathit{Sta}_X \mid \neg \mathrm{sound}_T(s)\})$.

In particular, T_0^* is the transition system in which all unsound states have been removed.

Of course, we can construct only sound ETSs simply by setting $\mathit{Err} = \emptyset$, but this is like denying that humans make errors. More useful is the following setting: start with an unsound untyped ETS, restrict the set of labels to those that pass the type-check of some type system, and then obtain a new ETS by restricting the transition relation to these labels; if this new error transition system is sound then one can say that the type system is sound w.r.t. the untyped ETS. We shall explore this later in more detail.

As well as soundness, we can define a notion of completeness for an ETS. For pointed transition systems, one could say that they are complete iff every state is reachable upto bisimilarity. Dually to the view that one can implement a partial function by a more complete function, it should be sufficient to reach a state that simulates the other one. In error transition systems, we are not interested in reaching error states or indeed unsound states. To sum up:

Definition 7
For arbitrary transition systems: a state s is called *complete* iff

$$\forall s_1 \in \mathit{Sta}.\, \exists s_1' \in \mathit{Sta}.\, (s \twoheadrightarrow s_1' \wedge s_1 \precapprox s_1').$$

It is called *bicomplete* iff

$$\forall s_1 \in \mathit{Sta}.\, \exists s_1' \in \mathit{Sta}.\, (s \twoheadrightarrow s_1' \wedge s_1 \approx s_1').$$

A pointed TS is (bi-) complete iff its initial state ι is (bi-) complete. An error transition system T is *strongly (bi-) complete* iff T_0 is (bi-) complete, and it is called (bi-) complete iff T^* is strongly (bi-) complete.

Any bicomplete ETS is complete, but the converse may not hold. We shall later see that there is a large class of ETS that fall into this gap. In some sense bicompleteness is therefore a more adequate notion.

The choice of $\approx$ and $\precapprox$ in the definition of bicompleteness and completeness instead of other notions of equivalence and approximation (e.g. trace equivalence) is motivated by the observation that the bisimulation equivalence on transition systems is a concept dual to congruence relations on Σ-algebras; for an elaborate treatment of this idea see [14]. In particular, we can view labelled transition systems as models of a programming language amongst which the fully abstract model is a terminal object; the latter is also given by quotienting any model by its largest bisimulation. Our notion of bicompleteness corresponds to asking whether the terminal model satisfies the formula $\forall s. \iota \twoheadrightarrow s$.

3 A Concrete Programming Language

We will now consider the pure λ-calculus as a concrete programming language, i.e. as an instance of the notion of pointed error transition system. The operational semantics is provided by the untyped λ-calculus with call-by-value evaluation. This choice (call-by-value) is somewhat arbitrary, but the questions we address and the answers we find are not significantly different from what we could do with other strategies.

We consider expressions over the abstract syntax as shown in table 1, for which we assume the usual notational conventions for the λ-calculus.

The metavariable e ranges over *expressions*, m over *matches* (a pair of a variable and an expression), and x ranges over a countably infinite set of variables. To define an operational semantics for this language we need a notion of value and environment, as defined in the right column in table 1. The metavariable v ranges over *values* which for this simple language are just closures – a closure is a pair of an environment and a match. E ranges over *environments*, finite lists

$$
\begin{array}{rcl}
e & ::= & x \mid (\lambda m) \mid (e\, e') \\
m & ::= & x.e
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & (E, m) \\
E & ::= & \Diamond \mid E[x \mapsto v] \\
r & ::= & v \mid \blacksquare
\end{array}
$$

Table 1: Abstract Syntax

$$
\frac{x \neq x' \quad E \vdash x \to r}{E[x' \mapsto v] \vdash x \to r}
\qquad
\frac{}{E[x \mapsto v] \vdash x \to v}
$$

$$
\frac{}{\Diamond \vdash x \to \blacksquare}
$$

$$
\frac{}{E \vdash \lambda m \to (E, m)}
$$

$$
\frac{E \vdash e \to (E', x.e'') \quad E \vdash e' \to v \quad E'[x \mapsto v] \vdash e'' \to r}{E \vdash (e\, e') \to r}
$$

$$
\frac{E \vdash e \to \blacksquare}{E \vdash (e\, e') \to \blacksquare}
\qquad
\frac{E \vdash e \to v \quad E \vdash e' \to \blacksquare}{E \vdash (e\, e') \to \blacksquare}
$$

Table 2: Expression Evaluation

of pairs of variables and values; r ranges over evaluation results, $\blacksquare$ is standing for an evaluation that has "gone wrong".

Table 2 defines the evaluation of expressions in the usual SOS style (see [10]).

We have three rules for variable access: (i) we skip irrelevant environment entries, (ii) we access value entries, and (iii) we report an error when we access an unbound variable, i.e. a variable in the empty environment. For this language this is the only elementary form of error. All other errors result from error propagation.

In the last rule we only propagate $\blacksquare$ when the first component reaches a value. Without that requirement, an implementation of this dynamic semantics would have to employ a parallel evaluation strategy (normally used for implementing angelic non-determinism [3]) for the evaluation of applications.

Having variable access as the only source for unsoundness is not enough to motivate a proper type system for preventing unsoundness and we will see later why. But even in that limited setting we can already state and prove soundness and completeness results by using techniques which can be adapted easily to more sophisticated languages.

Based on this traditional large-step operational semantics, we can define a corresponding ETS:

Definition 8
The ETS λ_v is defined as follows:

- $\mathit{Sta} = \mathit{Env} \cup \{\blacksquare\}$

- $\iota = \Diamond$
- $\mathit{Lab} = \mathit{Exp} \times \mathit{Var}$
- $E \xrightarrow{(e,x)} \blacksquare \iff E \vdash e \to \blacksquare$ and
  $E \xrightarrow{(e,x)} E' \iff \exists v \in \mathit{Val}.\, E \vdash e \to v \wedge E' = E[x \mapsto v]$
- $\mathit{Err} = \{\blacksquare\}$

Proposition 3 *The ETS λ_v is deterministic.*

The ETS λ_v is obviously unsound since the evaluation of a variable in $\Diamond$ results in $\blacksquare$. Since an environment only binds finitely many variables, *any* environment in λ_v is unsound; consequently, the notions of completeness and bicompleteness are not defined for λ_v since $(\lambda_v)^*_0$ has no states at all. As we shall see later, λ_v is strongly bicomplete.

In order to establish some general results for this untyped language we need a few auxiliary notions: context, free variable, etc.

For some meta-theoretic reasoning it is useful to have a notion of "expression with a hole". For our purposes it is sufficient to restrict this general idea of context to more specific ones.

Definition 9
Contexts are defined over the following abstract syntax:

$$C \quad ::= \quad \Box \mid ((\lambda x.C)\; e)$$

Given a context C and an expression e we write $C[e]$ for the expression defined as follows:

$$\begin{array}{rcl} \Box[e] & = & e \\ ((\lambda x.C)\; e')[e] & = & ((\lambda x.C[e])\; e') \end{array}$$

Definition 10
We define concatenation of environments as follows (infix notation):

$$\begin{array}{rcl} \Diamond +\!\!+ E & = & E \\ E[x \mapsto v] +\!\!+ E' & = & (E +\!\!+ E')[x \mapsto v] \end{array}$$

We assume the usual definition of free variable for expressions $\mathrm{FV}(e)$ and generalise it to values and environments as follows:

Definition 11
The *domain* of an environment E, $\mathit{Dom}\, E$, is a finite set of variables, defined as: $\mathit{Dom}\, \Diamond = \emptyset$ and $\mathit{Dom}\, E[x \mapsto v] = \{x\} \cup \mathit{Dom}\, E$. The *free variables* of values and environments are defined as follows:

$$\mathrm{FV}(E, m) = \mathrm{FV}(E) \cup (\mathrm{FV}(\lambda m) \setminus \mathit{Dom}\, E)$$

$$\begin{array}{l} \mathrm{FV}(\Diamond) = \emptyset \\ \mathrm{FV}(E[x \mapsto v]) = \mathrm{FV}(v) \cup \mathrm{FV}(E) \end{array}$$

Intuitively, values and environments should not contain any free variables. But we do not get this property for free. We will later establish a couple of results which will enable us to ignore non-closed values and environments for all intents and purposes.

For expressions, values, environments, etc. there is a notion of substitution, e.g. $E[e/x]$ is the environment obtained from E by replacing all free occurrences of x by the expression e. Substitution is restricted to the case where the substitute is a closed expression; this way there is no risk of name capture and substitution is purely syntactic.

4 Properties of Evaluation

To show certain properties of the simulation $\precapprox$ we first define a few other relations, some of which will turn out to be simulations or bisimulations.

Definition 12
We define a preorder $\precsim$ between values and a family of preorders $\overset{e}{\precsim}$ (indexed by expressions) between environments as the smallest preorders satisfying (on values):

$$(E, m) \precsim (E', m) \iff E \overset{\lambda m}{\precsim} E'$$

For environments: $E \overset{e}{\precsim} E'$ iff for all $x \in \mathrm{FV}(e)$: $E = E_1 \mathbin{+\!\!+} E_2[x \mapsto v]$ (with $x \notin \mathit{Dom}\, E_1$) implies that there are environments E_1', E_2' and a value v' with $E' = E_1' \mathbin{+\!\!+} E_2'[x \mapsto v']$ (and $x \notin \mathit{Dom}\, E_1'$) such that $v \precsim v'$.

Intuitively, $v \precsim v'$ holds if v and v' are the same except that closures in v' may have additional entries and may omit redundant ones; a binding is redundant if it binds a non-occurring variable.

Lemma 4 *If $E_1 \overset{e}{\precsim} E_2$ and $E_1 \vdash e \to v_1$ then there is a value v_2 such that $E_2 \vdash e \to v_2$ and $v_1 \precsim v_2$.*

Proof. By induction over the height of the derivation tree of $E_1 \vdash e \to v_1$.

For the evaluation of variables there are two (potentially) succeeding rules, so we have two cases to consider for deriving $E_1 \vdash x \to v_1$.

1. $E_1'[x \mapsto v_1] \vdash x \to v_1$. Then by definition of $\overset{x}{\precsim}$ we have $E_2 = E_2' \mathbin{+\!\!+} E_2''[x \mapsto v_2]$ with $x \notin \mathit{Dom}\, E_2'$ and $v_1 \precsim v_2$. From this we get $E_2 \vdash x \to v_2$.

2. Suppose $E_1'[x' \mapsto v] \vdash x \to v_1$ from $x \neq x'$ and $E_1' \vdash x \to v_1$. Clearly $E_1' \overset{x}{\precsim} E_1 \overset{x}{\precsim} E_2$ and the result follows from the induction hypothesis for $E_1' \vdash x \to v_1$.

For the evaluation of λ-bindings the result is immediate from the definitions of evaluation and $\precsim$.

Closure application: if we derive $E_1 \vdash (e\, e') \to v_1$ from $E_1 \vdash e \to (E_1', m)$, $m = x.e''$, $E_1 \vdash e' \to v$, and $E_1'[x \mapsto v] \vdash e'' \to v_1$ then we know by induction hypothesis on e that $E_2 \vdash e \to (E_2', m)$ and $(E_1', m) \precsim (E_2', m)$; by induction hypothesis on e' we have $E_2 \vdash e' \to v'$ and $v \precsim v'$. Now consider the environments $E_1'' = E_1'[x \mapsto v]$ and $E_2'' = E_2'[x \mapsto v']$. If we can show $E_1'' \overset{e''}{\precsim} E_2''$ then we get the result by induction hypothesis on $E_1'' \vdash e'' \to v_1$. Considering an arbitrary variable y free in e'', we either have $y = x$ for which the result is immediate from $v \precsim v'$, or $y \neq x$ for which the result easily follows from $(E_1', m) \precsim (E_2', m)$. $\square$

Definition 13
$E \precsim E'$ iff $\forall e.\, E \overset{e}{\precsim} E'$. $E \sim E'$ iff $E \precsim E' \wedge E' \precsim E$.

For the relation $\precsim$ we immediately get from lemma 4 and determinism of λ_v:

Corollary 5 *The relation $\precsim$ is a simulation and $\sim$ a bisimulation.*

The bisimulation $\sim$ is already a rather powerful tool. For example, it allows to show that the permutation and thinning of environment entries results in bisimilar environments. In very similar style we can establish other bisimulations, though we shall not go into the same detail. In particular, we want to get rid of free variables in values and environments.

Definition 14
We define relations $\smile$ on values and environments as the smallest equivalence relations such that for all variables x: $E \smile E[\bot/x]$ and $v \smile v[\bot/x]$ where $\bot$ is the expression $(\lambda y.y\, y)\,(\lambda y.y\, y)$.

Obviously, each $\smile$-equivalence class of values and environments has a closed member. Moreover, this member is unique.

Proposition 6

$$\forall v.\, \exists! v'.\, (v \smile v' \wedge FV(v') = \emptyset) \;\;\wedge\;\; \forall E.\, \exists! E'.\, (E \smile E' \wedge FV(E') = \emptyset)$$

The idea behind $\smile$ is that the $\bot$ expression is just as good as an unbound variable, because accessing an unbound variable during an evaluation results in $\blacksquare$ and accessing $\bot$ also fails to deliver a result. In other words:

Proposition 7 *The relation $\smile$ is a bisimulation.*

Remark: the formal proof is technically slightly different from the previous one, as one has to allow the replacement of expressions by equivalent ones, where the equivalence relation depends on the (domain of the) current environment.

Based on what we have so far we can define a new bisimulation equivalence $\fallingdotseq$ as $(\smile \cup \sim)^+$ which allows us to combine the pumping of closures with the replacement of free variables by $\bot$. Notice that we do not have to go through the operational semantics again to show that $\leftrightharpoons$ is indeed a bisimulation equivalence: it is a bisimulation, because bisimulations are closed under arbitrary union and under composition (and hence under transitive closure), and it is an equivalence relation because the transitive closure of any symmetric and reflexive relation is one.

Definition 15
We define inductively a pair of relations $\rightsquigarrow$ between values and expressions and between environments and contexts as follows:

$$(E, m) \rightsquigarrow C[\lambda m] \Leftarrow\ E \rightsquigarrow C$$

$$\Diamond \rightsquigarrow \Box$$

$$E[x \mapsto v] \rightsquigarrow C[(\lambda x.\Box)\, e] \Leftarrow\ v \rightsquigarrow e \wedge E \rightsquigarrow C$$

Read $v \rightsquigarrow e$ as "v is represented by e". Clearly, all values and environments have unique representants:

Proposition 8 $\forall v.\, \exists! e.\, v \rightsquigarrow e \wedge \forall E.\, \exists! C.\, E \rightsquigarrow C$

Moreover, closed values have closed representants:

Proposition 9 $\forall E, C, e.\, (E \rightsquigarrow C \ \supset\ FV(C[e]) \subseteq FV(E) \cup FV(e) \setminus \mathit{Dom}\, E)$ *and* $\forall v, e.\, (v \rightsquigarrow e \ \supset\ FV(v) \supseteq FV(e))$

The idea behind representants is that if e represents v and we evaluate e then we get v back. In this strict form this is (i) meaningless, because for an evaluation we have to give an environment as well, and (ii) not true (for any environment), because at least closures may differ in a certain way and free variables in v could spoil the result. So we have to consider values and environments modulo some bisimulation equivalence. What we can claim is the following:

Lemma 10 *Let v be a closed value and E be a closed environment. Suppose $v \rightsquigarrow e$ and $E \rightsquigarrow C$. Then we have*

1. $\forall E'.\, \exists v'.\, E' \vdash e \rightarrow v' \wedge v \leftrightharpoons v'$
2. $\forall v', e'.\, ((FV(e') \subseteq \mathit{Dom}\, E \wedge E \vdash e' \rightarrow v') \supset$
   $\forall E'.\, \exists v''.\, (E' \vdash C[e'] \rightarrow v'' \wedge v' \leftrightharpoons v''))$

Proof. By induction on the structure of v and E (notice that closedness is preserved, i.e. (E, m) is a closed value only if E is a closed environment, etc.).

Closures: let $v = (E_1, m)$ and $E_1 \rightsquigarrow C_1$; therefore $e = C_1[\lambda m]$. We have to show $E' \vdash e \rightarrow v'$ and $v \leftrightharpoons v'$ for some v'. Clearly $E_1 \vdash \lambda m \rightarrow (E_1, m)$, so we can apply the induction hypothesis for E_1 (second property) and obtain the result.

For environments, consider the empty environment: suppose $\Diamond \vdash a \to v'$ and $\mathrm{FV}(a) \subseteq \mathrm{FV}(\Diamond) = \emptyset$. Then for any E' we have $\Diamond \overset{a}{\sim} E'$ by definition of $\overset{a}{\sim}$ and hence by lemma 4 there is a value v'' such that $E' \vdash a \to v''$ and $v' \fallingdotseq v''$.

Non-empty environments: $E = E_1[x \mapsto v_1]$. Suppose $v_1 \leadsto e_1$ and $E_1 \leadsto C_1$. Take an arbitrary expression e' with $\mathrm{FV}(e') \subseteq \mathit{Dom}\, E$ such that $E \vdash e' \to v'$ for some v'. Consider the expression $e_0 = (\lambda x.e')\, e_1$. We have $E_1 \vdash e_0 \to v''$ iff we also have $E_1 \vdash e_1 \to v_1'$ and $E_1[x \mapsto v_1'] \vdash e' \to v''$. From the induction hypothesis on v_1 we know that there is a v_1' such that $E_1 \vdash e_1 \to v_1'$ and $v_1 \fallingdotseq v_1'$; this implies $E \fallingdotseq E'' = E_1[x \mapsto v_1']$ and thus there is a v'' with $E'' \vdash e' \to v''$ and $v' \fallingdotseq v''$. Now we can apply the induction hypothesis on E_1 and get for any E' a value v''' with $E' \vdash C_1[e_0] \to v''' \wedge v'' \fallingdotseq v'''$. By transitivity of $\fallingdotseq$ we have $v' \fallingdotseq v'''$. □

Now, from lemma 10 and proposition 6 it follows that for any value v there is an expression e the evaluation of which reproduces in any environment a value that bisimulates the original one.

Corollary 11 *Let E be a closed environment with $E \leadsto C$.*
If $E +\!\!+ E' \vdash e \to v$ then there is a $v' \fallingdotseq v$ such that $E' \vdash C[e] \to v'$. Conversely, if $E' \vdash C[e] \to v'$ then there is a value $v \fallingdotseq v'$ such that $E +\!\!+ E' \vdash e \to v$.

Proof. Follows easily from lemma 10 by induction on the length of E. For example, the induction step for environments goes as follows (we omit the bisimilarity argument for the values; e_v is the representant of v):
$E_0[x \mapsto v] +\!\!+ E' \vdash e \to v_0 \iff E_0 +\!\!+ E' \vdash (\lambda x.e)\, e_v \to v_1 \iff$
$E' \vdash C_0[(\lambda x.e)\, e_v] \to v_2 \iff E' \vdash C[e] \to v_2$ □

Because λ_v is deterministic we already know from lemma 2 that $\sqsubseteq$ is a simulation and $\equiv$ a bisimulation. From the properties we have established about $\fallingdotseq$ we can show something stronger: environment concatenation $+\!\!+$ is monotonic w.r.t. to $\sqsubseteq$, and therefore the trace order on environments is directly reflected on values.

Theorem 12 *The ETS λ_v is strongly bicomplete.*

Proof. We have to show that any environment is reachable from $\Diamond$ upto $\approx$. We show the stronger property that any $E[x \mapsto v]$ is reachable in a single step from E, the theorem then follows by induction on the length of the environment. The value v is bisimilar (via $\fallingdotseq$) to some closed value v' which has a representative $v' \leadsto e$. Hence by lemma 10 we have $E \vdash e \to v''$ with $v \fallingdotseq v''$ and so by definition of transition $E \overset{(e,x)}{\longrightarrow} E[x \mapsto v'']$ and by definition of $\fallingdotseq$ on values: $E[x \mapsto v] \fallingdotseq E[x \mapsto v'']$ which implies $E[x \mapsto v] \approx E[x \mapsto v'']$. □

5 Abstract Type Systems

What is common to all type systems? Typically, they come with a ternary relation $B \vdash e : \tau$ which states that an expression e has type τ in a type environment B. Having this type τ can be seen as an assertion that it is safe to evaluate the expression e in any environment E that "fits" B. We can think of the type as standing for a class of values and the type environment as standing for a class of (value) environments. The "fitting" relation is a bit vague in general, but in any case the empty environment should fit the empty type environment. Therefore, the ingredients of a type system are similar (in an informal sense) to the ingredients of the programming language which it is attached to.

The above argument is somewhat biased towards functional programming languages. We are looking for the general scenario, i.e. type systems for arbitrary pointed transition systems. What is a type system in this abstract setting? Generalising from the observation we just made for functional programming, a type system for a pointed TS should be another pointed TS. This is a bit like taking the slogan "typing is an abstract interpretation" and turning it around: "any abstract interpretation induces a type system".

To link the type system with the operational semantics we only need one thing: a link between the syntaxes of both systems.

Definition 16
A *type system* is a structure $(A, B, \doteq)$ such that A and B are pointed transition systems and $\doteq \subseteq Lab_A \times Lab_B$.

In the following we will concentrate on type systems with $Lab_A = Lab_B$ and where $\doteq$ is the equality on labels.

What does it mean to have a type system? The original idea from functional programming was to restrict the set of expressions to well-typed ones. A well-typed expression is simply an expression that has a type. Dually, environments have types: the corresponding type environments. The general picture is that the states of the type system serve as types for the states of the other transition system.

Definition 17
Given a type system $(A, B, \doteq)$ we define the associated *untyped transition system* $A : B$ as the pointed TS given as:

- the set of states $Sta_A \times Sta_B$,
- the set of labels $Lab_A \times Lab_B$,
- the initial state (ι_A, ι_B),
- the transition relation:
$(X, Y) \xrightarrow{(x,y)} (X', Y') \iff x \doteq y \wedge X \xrightarrow{x} X' \wedge Y \xrightarrow{y} Y'$.

In other words, the associated untyped transition system is just the pointwise product — except for the labels which are synchronised via the label relation $\doteq$. Notice that the construction is symmetric: the transition system we get from imposing A as a type system for B is isomorphic to the one we get from switching their rôles.

We do not have yet a notion of soundness or completeness — for this we need again error states.

Definition 18
A type system for an ETS $E = (T, Err)$ is a type system $(T, A, \doteq)$. Its associated error transition system $E : A$ is $(T : A, Err \times Sta_A)$.

The idea is that we reach an error whenever we reach an error in the underlying untyped world. Instead of the above one could use ETSs themselves to type ETSs, with the idea that error states in the type system represent type errors. However, this would not change the break of symmetry.

We can lift the notions of soundness etc. from ETSs to ETSs with type systems: a type system A for an ETS is called sound (complete) iff the associated ETS is sound (complete). One can check that this gives indeed the usual notion of soundness. In particular, for λ_v: a type system for λ_v is sound iff no expression that type-checks in the initial type environment gives a run-time error in the initial environment of the underlying untyped language.

Establishing the similarity or bisimilarity of states in the typed system can be arbitrarily tricky. States that could be distinguished in the original system might become indistinguishable due to the sudden lack of observers (labels that pass the type check). Indistinguishability is preserved though:

Proposition 13 *Let A and B be pointed transition systems. Let $s_1 \precapprox_A s_2$ and $b_1 \precapprox_B b_2$. Then $(s_1, b_1) \precapprox (s_2, b_2)$ in $A \doteq B$ for any $\doteq$.*

6 Concrete Type Systems

What would a sound and complete type system for λ_v look like? We restrict our attention to the situations in which the labels are equal and $\doteq$ is the identity relation. A slightly unusual type system is given by the following pointed transition system which we call λ^0:

- states: finite sets of variables with $\emptyset$ as initial state,
- labels: as for λ_v, i.e. pairs of expressions and variables,
- transitions: $M \xrightarrow{(e,x)} N \iff N = M \cup \{x\} \wedge \mathrm{FV}(e) \subseteq M$.

Since the only elementary error in λ_v is to access an unaccounted variable, it should be intuitively clear that λ^0 is sound for λ_v. Indeed:

Theorem 14 $\lambda_v : \lambda^0$ *is sound.*

Proof. We prove by induction on the derivation trees the stronger invariant that whenever $E \vdash e \to r$ and $\mathrm{FV}(e) \subseteq \mathit{Dom}\, E$ and $\mathrm{FV}(E) = \emptyset$ then $r \neq \blacksquare$ and $\mathrm{FV}(r) = \emptyset$. Clearly $\mathit{Dom}\, E = M$ in any reachable state (E, M) and hence it follows that $\mathrm{FV}(E) = \emptyset$ which means that $\blacksquare$ is unreachable. □

What about completeness? This is not such a trivial question as one first might think, because the restriction to closed terms affects extensional equality (see [1, 8]). Still, the pointed TS λ^0 is bicomplete for λ_v:

Theorem 15 $\lambda_v : \lambda^0$ *is bicomplete.*

Proof. Take an arbitrary non-error state (E, M). We have to show that it is reachable (up to $\approx$) if it is sound.

Suppose $x \in M \setminus \mathit{Dom}\, E$: in this case the state is unsound since we can reach $\blacksquare$ by the transition $(E, M) \xrightarrow{(x,x)} \blacksquare$. Therefore we can assume $M \subseteq \mathit{Dom}\, E$.

Define $E \downharpoonleft M$ to be the restriction of E to variables occurring in M. By an argument very similar to the proof of lemma 4 we have $E \downharpoonleft M \overset{e}{\sim} E$ for all e with $\mathrm{FV}(e) \subseteq M$. This implies $(E, M) \approx (E \downharpoonleft M, M)$. By strong bicompleteness of λ_v (theorem 12) the environment $E \downharpoonleft M$ is reachable up to $\approx$. Suppose $E \downharpoonleft M \approx E'$ and $\Diamond \xrightarrow{w} E'$. Lemma 13 implies that $(E', M) \approx (E, M)$. The domains of E' and $E \downharpoonleft M$ are equal to each other (by bisimilarity) and to M (by the earlier assumption). Moreover, the concrete w obtained from the proof of theorem 12 only consists of closed expressions. But for any closed expression e we have the λ^0 transition $M \xrightarrow{(e,x)} M \cup \{x\}$ and thus (E', M) is reachable. □

If we consider more traditional type systems for the λ-calculus, for example λ^τ or other systems of the λ-cube (see [2]), establishing soundness of these type systems for λ_v is possible through proving a subject reduction property. These systems typically exhibit the difference between completeness and bicompleteness. The transition system for λ^τ simply uses types as values and modus ponens for application.

Theorem 16 $\lambda_v : \lambda^\tau$ *is complete but not bicomplete.*

Proof. Suppose (E, B) is a sound state. Any reachable state of the form (E', B) simulates (E, B) because all expressions that type-check in B are strongly normalising in E' (by the SN property of λ^τ). If there is no reachable state of the form (E', B) then B contains bindings to uninhabited types. We can replace all occurrences of type 0 in B by $0 \to 0$ getting a new type environment B'; the state (E, B) is simulated by (E, B'), in particular all expressions that type-check in B also type-check in B'. Moreover, all types in B' are inhabited, so that we can find some reachable (E', B') which by the previous argument simulates (E, B') and (by transitivity) (E, B).

Bicompleteness fails, because we could give a variable x the type $(0 \to 0) \to (0 \to 0)$ and bind it to $(\Diamond, z.\bot)$ where $\bot$ is some non-terminating expression. This is a sound pair, but it is not bisimilar to any reachable state. □

One might regard the completeness argument in the proof as unsatisfactory as we simply replaced all troublesome types by other types. The reason we could do this is that types are not directly observable in λ^τ, they are part of the states and not part of the transitions. To forbid the change of types one could modify transitions in λ^τ from (e, x) to (e, x, t) where t is the type of x and e. This change would ensure that types are preserved by simulation and it would imply the incompleteness of the corresponding typed transition system.

The lack of bicompleteness of $\lambda_v : \lambda^\tau$ due to the failure of expressing non-termination may seem innocuous, but it is a bit more serious than one might expect. With an argument very similar to the one used for the proof of the completeness claim one can show that any two reachable states (E, B) and (E', B') are bisimilar if $B = B'$. The reachable part of the transition system $\lambda_v : \lambda^\tau$ (divided by bisimilarity) is like the minimal model of λ^τ in which all carrier sets are either singletons or empty, depending on whether the proposition corresponding to the type is true or not. In particular, we cannot distinguish different Church-numerals. Even if we added a zero-tester and a predecessor, different Church-numerals would still be indistinguishable because we still have a strongly normalising language. The further addition of a value that triggers non-termination (like the counterexample to bicompleteness) changes all that: non-termination of a λ_v evaluation is observed at the transition system level as the failure to make a transition with a certain label and thus it affects bisimilarity.

In other words, the completeness question is more interesting for type systems that allow non-termination, e.g. $\lambda*$ or PCF. PCF is essentially λ^τ extended by natural numbers and fixpoints, see [9]. Using call by value evaluation we assume that it is an error to evaluate the PCF term (Succ x) if x is not bound to a natural number. This induces a partial logical equivalence relation on all values (see [7]) and a PCF state (E, B) is sound iff the environment E (restricted to the domain of B) is logically related to itself w.r.t. B. From this we can deduce the bicompleteness of PCF:

Theorem 17 *PCF is bicomplete.*

Proof. Let (E, B) be a sound (but unreachable) PCF state. We want to construct a reachable state (E', B) which is bisimilar to (E, B). We do this pointwise for all variables bound in B. We can do this by encoding the syntax of PCF in PCF and writing the obvious evaluation functions for evaluating gödelised open expressions in a given (typed) environment. Notice that we need one function for each combination of types for the environment and the expected type for the expression, but for each concrete typed value finitely many such functions suffice. □

One might expect that the argument in the proof of theorem 17 carries over to arbitrary (Turing-complete) programming languages. However, it is not always possible to express an evaluation function *within* the language, i.e. traversing the structure of an expression may already violate the typing rules. In particular, Standard ML [6] is incomplete: for recursive datatypes which “mutate” during recursion it is not possible to fully traverse their data, example:

```
datatype 'a opent =  ZZ | SS of ('a t) t | Var of int
datatype 'a t = Z | S of ('a t) t
fun eval ZZ env = Z
 |  eval (SS x) env = S (eval x env)
 |  eval (Var n) (v::xs) =
        if n=0 then v else eval (Var (n-1)) xs
```

The function `eval` can be seen as evaluating (a subset of the) encoded open expressions of type `'a t` in a given environment. It is sound, but its definition does not type-check in SML and it is not possible to define this function by other means either [4].

7 Conclusion and Related Work

We have defined a general notion of what it means to be a type system for an arbitrary transition system, and express what it means for such a system to be sound and/or complete. So far, the literature has focussed on looking at particular notions of soundness of particular programming languages, e.g. [15] for Standard ML.

Related to the work presented here is a recent paper by Puntigam [12]. Puntigam aims at types for object oriented languages and uses the trace semantics for typing. In several ways this is more specific than what we suggest here, but it is based on the same fundamental idea of linking labelled transitions with types. While Puntigam has a different and more specific objective but uses similar methods, van Raamsdonk and Severi [13] also aim at typing for transition systems but do it very differently; still, they also use states as types for states.

We showed the completeness and lack of bicompleteness of the type system λ^τ. It should be clear that the proof of the latter easily generalises to all type systems in the λ-cube as they admit the same counter-examples, but it is less clear and indeed dubious whether the completeness argument goes through as well. By construction, reachable environments are inhabited and therefore they are tautologies when we interpret them as propositions using the Curry-Howard isomorphism. Conversely, any environment that has a tautological type environment is reachable (up to simulation). This holds for all strongly normalising type systems for the λ-calculus. We could show the completeness of λ^τ by exhibiting for any non-tautological type environment a tautological one that type-checks the same (and more) expressions. This trick does not carry over to more expressive type systems than λ^τ.

We also showed the bicompleteness of PCF. The proof of this property can apparently be adapted to many other Turing-complete programming languages; still, Standard ML is (surprisingly) incomplete.

References

[1] Hendrik P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics.* North-Holland, 1984.

[2] Hendrik P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol.2*, pages 117–309. Oxford Science Publications, 1992.

[3] D. Gries and D. Jacobs. General correctness: a unification of partial and total correctness. *Acta Informatica*, 22:67–83, 1985.

[4] Stefan Kahrs. Limits of ML-definability. (submitted), 1996.

[5] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[6] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[7] J.C. Mitchell. Type systems for programming languages. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 8, pages 365–458. Elsevier, 1990.

[8] Gordon Plotkin. The λ-calculus is ω-incomplete. *The Journal of Symbolic Logic*, 39:313–317, 1974.

[9] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[10] Gordon Plotkin. A stuctural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[11] Sally Popkorn. *First Steps in Modal Logic.* Cambridge University Press, 1994.

[12] Franz Puntigam. Types for active objects based on trace semantics. In *Proceedings FMOODS'96*, pages 5–20. IFIP WG 6.1, 1996.

[13] Femke van Raamsdonk. *Confluence and Normalisation for Higher-Order Rewriting.* PhD thesis, Vrije Universiteit Amsterdam, 1996.

[14] J.J.M.M. Rutten. A calculus of transition systems. In Alban Ponse, Maarten de Rijke, and Yde Venema, editors, *Modal Logic and Process Algebra*, pages 231–256. CSLI pubplications, 1995.

[15] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, 1988. CST-52-88.

[16] Glynn Wynskel and Mogens Nielsen. Models for concurrency. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 1–148. Oxford Science Publications, 1995.