Static Analysis of Actors: From Type systems to Abstract Interpretation

Pierre-Loïc Garoche and Marc Pantel and Xavier Thirioux

IRIT, Toulouse
{garoche,pantel,thirioux}@enseeiht.fr

Abstract

We focus on the actor model, as it eases the definition of concurrent programs with non-uniform behaviors. Our static analyses of such a model were previously based on type systems. Due to limitations in terms of expressivity as well as precision, we moved to a new abstract interpretation framework and intended to replace our type systems by abstract domains behaving the same way. Our new analyses are now able to verify most of our properties of interest, as well as new ones. This paper focuses on the translation process, relating the pros and cons of both techniques from a practical standpoint. We show that our type systems could be quite naturally embedded in an abstract interpretation framework, which finally seems more promising.

Key words: Type systems, abstract interpretation, flow analysis, linearity analysis.

1 Introduction – Context

Our work is focused on applying formal methods to improve concurrent object oriented programming. To obtain widely usable tools, we have chosen to use the actor model proposed by Hewitt [16] and developed by Agha [1]. This model is based on a network of autonomous and cooperative agents, called actors, which encapsulate data and programs, communicating using an asynchronous point to point protocol. An actor stores each received message in a queue and when idle, processes the first message it can handle. Besides those conventions, an actor can dynamically change its interface, i.e. the set of messages it may handle, yielding a more accurate programming model. This model, also known as concurrent objects with non-uniform behavior, has been adopted by the telecommunication industry for the Open Distributed Computing framework (ITU X901-X904) and the Object Description Language (TINA-C extension of OMG IDL with multiple interfaces). Until now, we have been designing several analyses for an actor model, all of which based

on type systems. Our main objective is to detect in an accurate way typical flaws of distributed applications, like for instance communication deadlock or non linearity (i.e. the fact that several distributed actors have the same address). One drawback of type-based analysis is its data-flow flavor. Although control-flow analysis can be mimicked with sophisticated encodings [21], abstract interpretation seems to be more adequate. It has been recently applied with success to concurrent and distributed programming with the works by VENET [23] and later FERET [13,14]. Due to limitations of our previous attempts, we decided to move to the framework of abstract interpretation, whose tools and ideas have significantly grown in maturity and are being widely used in industrial contexts, or are on the verge of being so. We now investigate these techniques in order to capture more complex properties, such as detection of orphan messages, that is messages sent to an actor which will not handle them. Moreover, we may also capture new properties, especially dedicated to control of resources' usage.

2 An actor dedicated process calculus: CAP

In order to ease the definition of static analyses for actor based systems, we have defined the CAP primitive actor calculus [7], by merging asynchronous π-calculus together with ABADI and CARDELLI's Primitive Object Calculus. We here describe CAP and then define the property which we will focus on.

2.1 Syntax and semantics

Let $\mathscr{N}$ and $\mathscr{V}$ be respectively the infinite sets of actor names and variables. Let $\mathscr{L}_m$ be a set of message labels. Let $\mathscr{L}_p$ and $\mathscr{L}_n$ be respectively the sets of program point and name labels. $\mathscr{L}_b$ denotes the subset of $\mathscr{L}_p$ of restriction program points. Finally, we define $\mathscr{L} = \mathscr{L}_p \cup \mathscr{L}_n$. The syntax of configurations is described as follows:

$$C ::= 0 \mid \nu a^{\alpha}\ C \mid C \parallel C \mid a \rhd^{l} P \mid a \lhd^{l} m(\widetilde{P})$$
$$P ::= x \qquad \mid \qquad [m_i^{l_i}(\widetilde{Var}) = \zeta(e_i^{\alpha_i}, s_i)C_i]^{i \in I}$$

Configurations can be respectively: an empty process; a creation of an actor's address; parallel execution; an attachment of a behavior defined by P to an actor on address a; and finally, a message sent to an address a with arguments $\widetilde{P}$. Program points define messages, behaviors' installation or external choices between behaviors. The name binders are: name restrictions $(\nu a^{\alpha})C$, $\zeta(e_i^{\alpha_i}, s_i)$ operators and message labels for the parameters $\widetilde{x_i}$ in behavior terms $[m_i^{l_i}(\widetilde{x_i}) = \zeta(e_i^{\alpha_i}, s_i)C_i]^{i \in I}$. Therefore, the occurrences of a in C, $\widetilde{x_i}$ in $\zeta(e_i^{\alpha_i}, s_i)C_i$ and e_i and s_i in C_i are bound. The ζ operator is our reflexivity operator, it catches both address and behavior of its actor and allows to re-use

them in the behavior. We denote by $\mathcal{FV}(C)$ the set of free variables in C. The initial semantics of CAP is defined à la Milner, by both a standard congruence relation which is omitted here for the sake of brevity, and a transition rule:

$$\frac{T = [m_i^{l_i}(\widetilde{x_i}) = \zeta(e_i^{\alpha_i}, s_i)C_i]^{i\in I} \quad length(\widetilde{y}) = length(\widetilde{x_k}) \quad k \in I}{a \rhd^l T \;||\; a \lhd^{l'} m_k(\widetilde{y}) \xrightarrow{(l,l_k,l')} C_k[e_k \leftarrow a, s_k \leftarrow T, \widetilde{x_k} \leftarrow \widetilde{y}]}$$

2.2 An Example: the Duplicating Server

The following example $FullEx$ illustrates the expressivity of CAP. We first define a behavior $Serv$, able to handle a message m with one argument c. It replicates itself and sends a *reply* message to c. When an actor with behavior $Serv$ receives a message $reify$, it sends both *ego* and $self$ to a duplicating server. Such a server creates two new addresses, associated with the behavior $Serv$ received in the message. Then it associates to the received address the behavior Aux. When receiving a message, the actor with behavior Aux sends it to the duplicated actors which will then send their reply to a new actor created to forward the first answer to the original recipient.

$$\begin{aligned}
Serv &= [m(c) = \zeta(e,s)(e \rhd s \;||\; c \lhd reply()),\\
&\qquad reify(d) = \zeta(e,s)(d \lhd server(e,s))]\\
DupServ &= [server(ego, self) = \zeta(e,s)\\
&\qquad (e \rhd s \;||\; \nu a, b(a \rhd self \;||\; b \rhd self \;||\; ego \rhd Aux(a,b)))]\\
Aux(a,b) &= [m(c) = \zeta(e,s)\\
&\qquad (e \rhd s \;||\; \nu f(a \lhd m(f) \;||\; b \lhd m(f) \;||\; f \rhd Join(c)))]\\
Join(c) &= [reply() = \zeta(e,s)(c \lhd reply() \;||\; c \rhd [reply() = 0])]\\
FullEx &= \nu s, d, c(s \rhd Serv \;||\; d \rhd DupServ \;||\; s \lhd reify() \;||\; s \lhd m(c))
\end{aligned}$$

2.3 The property of interest: Linearity

Several properties were checked for CAP terms using type inference systems relying on set and multi-set constraints: message arity and trivial orphans (messages which can never be taken into account) [5]; linearity for actors [8]; orphan messages [6,9].

For the purpose of this paper, we will only consider the linearity analysis as it is complex enough to relate both approaches fairly. Its main purpose is to ensure that at any time only one behavior is reading messages from an actor mailbox. Such an analysis is not trivial in CAP since an address may be dynamically associated to many behaviors in its lifetime. A lot of work has been done for checking this kind of properties using types (see for example

Pierce *et al.* in [22], Kobayashi in [18], Boudol in [4] or Colaço *et al.* in [8]). We do not propose here a new type system but focus on comparing our previous proposal with more recent work on abstract interpretation.

3 Static analysis based on type inference

We now recall an implicit type system dedicated to linearity analysis. We use constrained types using the $HM(X)$ form (*cf.* [20]) and then solve accumulated constraints using algorithms inspired from those of Aiken and Wimmers [2] to check the property. Sub-typing mechanisms express behavioral properties of terms. We first define our types and sub-types, then give the typing rules. Here we do not describe the constraint solving algorithm and invite the attentive reader to refer to [10].

3.1 Types and sub-types

Our types and sub-typing relations are as follows, where π, χ, α, β and ι respectively denote types of processes, communicable values, addresses, behaviors and interfaces (sets of messages) and finally μ denotes usage modes:

$$\begin{array}{lll} \pi ::= \wp & \chi ::= \alpha \mid \beta & \alpha ::= \rho \overset{\mu}{\triangleright} \iota \\ \beta ::= \triangleright\iota & \iota, \rho ::= \langle m_i(\tilde{\chi_i})\rangle^{i\in I} & \mu ::= \circ \mid \bullet \mid \top \end{array}$$

$$\rho' \overset{\mu'}{\triangleright} \iota' \subseteq \rho \overset{\mu}{\triangleright} \iota \;\equiv\; \iota' \subseteq \iota \wedge \rho \subseteq \rho' \wedge \mu' = \mu \qquad \iota'\triangleright \subseteq \iota\triangleright \;\equiv\; \iota \subseteq \iota'$$

The kind of abstraction provided by these types is twofold: First, we have sets of messages that may be sent to addresses or processed by behaviors. Second, we have usage modes denoting respectively whether a given address corresponds to zero, one or more processes attached to that address. The last case corresponds to a potentially non-linear system, therefore rejected by our analysis, as unsafe addresses are given the usage mode $\top$. As modes are merely finite counters, we define addition on these modes with its obvious meaning, and extend this sum component-wise to deal with environments.

3.2 The type system

3.2.1 Typing rules

$$\text{(Behavior)}\frac{\begin{array}{c} E \text{ is unlimited and } \forall i \in I \\ E, \tilde{x_i} : \tilde{\chi_i}, s_i : \triangleright\iota, e_i : \rho_i \overset{\bullet}{\triangleright} \iota_i \vdash C_i : \wp \end{array}}{E \vdash [m_i(\tilde{x_i}) = \zeta(e_i, s_i)C_i]^{i\in I} : \triangleright\iota} \left(\begin{array}{c} \iota \cap \langle m_i(\ldots)\rangle \subseteq \langle m_i(\tilde{\chi_i})\rangle \\ \triangleright\iota_i \subseteq \triangleright\iota \\ \rho_i \subseteq \iota_i \end{array}\right)$$

The sub-typing constraint states that a message sent to an address attached to this behavior should be accepted by one of the m_i's, or by one of the behaviors attached to some e_i. The «E is unlimited» constraint states that when an address represented by a free variable inside a behavior term is attached to some behavior, that address should be considered unsafe, because in this case we cannot estimate the number of times this behavior term may be used. In an unlimited environment, usage modes are either $\circ$ or $\top$. Such a free variable may nevertheless be used as a message target without any restriction. The usage mode of e_i states that we can again attach one behavior to this address, since the communication of a message m_i leaves it unattached.

$$\text{(Actor)}\frac{E, a : \rho \overset{\circ}{\triangleright} \iota \vdash T : \triangleright\iota'}{E, a : \rho \overset{\bullet}{\triangleright} \iota \vdash a \triangleright T : \wp} \quad \left(\triangleright\,\iota' \subseteq \triangleright\iota\right) \qquad \text{(Null)}\frac{E \text{ is strictly linear}}{E \vdash 0 : \wp}$$

In the actor typing rule, the sub-typing constraint states that each behavior assigned to an address should be compatible with the type of this address. To be assigned, an address should be not already attached, and this usage mode naturally changes when typing the attached behavior. As for the null process rule, we impose that the environment E be strictly linear, i.e. it cannot contain any unattached or unsafe address.

$$\text{(Message)}\frac{E \vdash a : \rho \overset{\circ}{\triangleright} \iota \quad E_k \vdash T_k : \chi_k \;\; (\forall k \in [1 \cdots n])}{E + \sum_{k\in[1\cdots n]} E_k \vdash a \triangleleft m(T_1 \dots T_n) : \wp} \quad \left(\langle m(\chi_1 \dots \chi_n)\rangle \subseteq \rho\right)$$

The message typing rule constraint imposes that a communicated value is understood by an installed target address. Moreover, the linearity condition forces the typing rule to distribute the unattached addresses (seen as resources) among the parameters of this message and the target address. Therefore, this typing occurs in a linear sum of environments.

$$\text{(Parallel)}\frac{E_C \vdash C : \wp \qquad E_D \vdash D : \wp}{E_C + E_D \vdash C \parallel D : \wp} \qquad \text{(Restriction)}\frac{E, a : \rho \overset{\bullet}{\triangleright} \iota \vdash C : \wp}{E \vdash \nu a\, C : \wp} \quad \left(\rho \subseteq \iota\right)$$

Similarly to the previous rule, the sum of environments in the parallel typing rule expresses that unattached addresses should be shared between the two sub-terms. The name restriction typing rule creates a name available for behavior assignments. Finally, to these core rules, we must add rules about environment fetching, which are standard and therefore omitted.

3.2.2 Properties of our type system

We state here some properties, without any proof.

Proposition 3.1 (Typability of subterms) *If $E \vdash A : \tau$ and B is a subterm of A, then $\exists E'$ such that $E' \vdash B : \tau'$.*

Proposition 3.2 (Subject reduction) *If $E \vdash A : \tau$ and $A \longrightarrow A'$, then $\exists E'$ such that $E' \vdash A' : \tau'$.*

Proposition 3.3 (Property preservation) *If a term A is well-typed, the repeated use of the transition rule will never produce any communication errors, i.e. message arity or sort (between address and behavior) discrepancies. Moreover, the configurations reachable from A are linear if A is linear.*

3.3 Limits of our type system

The first point is that our approximation of control-flow yields a great loss in precision and may generate many spurious errors. It is not at all obvious how we could alleviate this problem by using a more refined type system while keeping not too complex type structures. For instance, the use of conditional types as advocated by Aiken and Wimmers in [3] leads to more precise results, as these types help in representing the control-flow more precisely, but at the cost of an exponentially worse complexity yet.

Another point is that each time we devise a new type based analysis, we must show that it is sound using a structural induction based subject reduction proof which greatly depends on the type structure.

We moved to abstract interpretation since this framework indeed allows to factor a great part of the soundness proofs while taking into account both data and control-flow in a precise way. The next part will show how we managed to define the same linearity analysis.

4 Static analysis by abstract interpretation

In this section, we describe how to verify the linearity property on CAP terms in the framework of Abstract Interpretation. We use the Meta framework defined by Feret in [14] which aims at describing the non-standard semantics of models for mobile systems. We will first explain how CAP can be expressed in such a Meta semantics, also called the non-standard semantics. We then approximate the collecting semantics of a term by the use of abstract domains. The abstract interpretation framework drives the way we represent the linearity property about the analyzed term and the way we compute a sound upper approximation of this property, abstracting the whole set of reachable configurations from the initial term.

4.1 Non-standard semantics

Here we encode CAP into a simplified version of Meta. In this semantics, a configuration of a system is a set of threads. Each thread t is a triple defined as $t = (p, id, E) \in \mathscr{L}_p \times \mathscr{M} \times (\mathscr{V} \to (\mathscr{L} \times \mathscr{M}))$ where p is the program point representing the thread in the CAP term, id is its history marker and E its

environment. E is a partial map from a variable to a pair $(value, marker)$, which domain, also called interface, is determined by p. Each marker is a word composed of program points representing the history of transitions which led to the creation of values or threads. It is required in order to distinguish recursive instances of a value or thread.

We will describe some primitives that allow us to define the non-standard semantics, then, briefly, we show how to compute transitions in this semantics.

4.1.1 Partial interactions

Each program point is associated to a set of partial interactions, which drive the transition rules. We have one to represent a message, one to represent a syntactically defined actor, one to represent a dynamically defined actor and one to describe a behavior. The last one is not consumed when interacting, it acts like a definition.

A partial interaction is given by a tuple $(s, (param_i), (bound_i), cont)$ where: $s \in \mathscr{A}$ is a partial interaction name; $(param_i) \in \mathscr{V}^\star$ is a finite sequence of variables (X_i); $(bound_i) \in \mathscr{V}^\star$ is a finite sequence of distinct variables (Y_i); $cont \in \wp(\mathscr{L}_p \times (\mathscr{V} \to \mathscr{L}))$ is a syntactic continuation.

We use both sequences $(param_i)$ and $(bound_i)$ to compute value passing. The set $cont$ determines which threads have to be inserted in the system.

4.1.2 Abstract syntax extraction

We now define abstract syntax extraction primitives. We map each program point labeled $l \in \mathscr{L}_p$ to a set of partial interactions and to an interface:

- Each program point label $a \rhd^l [m_i^{l_i}(\widetilde{x_i}) = \zeta(e_i^{\alpha_i}, s_i)C_i]^{i \in I}$ is associated to the interface $\{a\}$ and to the following set of partial interactions:

$$\bigcup_{i \in I} \{(static\_actor_n, [a, m_i], [e_i, s_i, \widetilde{x_i}], \beta(C_i, [e_i \mapsto \alpha_i]))\}$$

  We add a link between program points l_i and l to allow the use of the behavior later in a dynamic actor (see 4.1.3).
- Each program point label $a \rhd^l x$ is associated to the interface $\{a, x\}$ and to the partial interactions $\{(dynamic\_actor, [a, x], \emptyset, \emptyset)\}$
- Each program point label $a \lhd^l m(\widetilde{P})$ is associated to the interface $\{a\} \cup \mathcal{FV}(\widetilde{P})$ and to the partial interactions $\{(message_n, [a, m, \widetilde{P}], \emptyset, \emptyset)\}$
- Each program point label l_i corresponding to a particular behavior of an actor *i.e.* «$m_i^{l_i}(\widetilde{x}) = \zeta(e_i^{\alpha_i}, s_i)C_i$» is associated to the interface $\mathcal{FV}(C_i) \setminus \{e_i, s_i\}$ and to the partial interactions $\{(behavior_n, [m_i], [e_i, s_i, \widetilde{x}], \beta(C_i, [e_i \mapsto \alpha_i]))\}$

The initial state for a term $\mathscr{S}$, i.e. a set of potential continuations in $\wp(\wp(\mathscr{L}_p \times (\mathscr{V} \to \mathscr{L})))$, is defined as $init_s = \beta(\mathscr{S}, \emptyset)$, where the syntax extraction func-

tion β is defined inductively over the syntactic continuation:

$$\begin{aligned}
\beta(\nu a^{\alpha} C, E_s) &= \beta(C, E_s[a \mapsto \alpha]) \\
\beta(a \rhd^{l} [m_i^{l_i}(\widetilde{x_i}) = \ldots], E_s) &= \{\{(l, E_s)\} \cup \{(l_i, E_s) \mid i \in I\}\} \\
\beta(a \rhd^{l} b, E_s) &= \{\{(l, E_s)\}\} \\
\beta(a \lhd^{l} m(\widetilde{P}), E_s) &= \{\{(l, E_s)\}\} \\
\beta(C_1 || C_2, E_s) &= \{c_1 \cup c_2 \mid c_1 \in \beta(C_1, E_s), c_2 \in \beta(C_2, E_s)\} \\
\beta(0, E_s) &= \{\emptyset\}
\end{aligned}$$

4.1.3 Formal rules

We now define the formal rules which drive the interaction between threads. In the case of CAP, we have two rules that describe an actor handling a message, depending on the kind of actor we have, a static or a dynamic one.

In the following, the i-th parameter, the j-th bounded variable, and the identity of the k-th partial interaction are respectively denoted by X_i^k, Y_j^k and I^k. We define the endomorphism $behavior\_set$ on the set $\mathscr{L}_p \times \mathscr{M}$ as follows: $(p, m) \mapsto (p', m)$ where p is a behavior program point and p' is the program point where p has been syntactically defined. As an example, in the term $\nu^{\alpha} a, a \rhd^{1} [m^2() = \zeta(e^{\beta}, s)C]$, we have $behavior\_set(2, m) = (1, m)$.

There are two rules, one denoting the communication with a static actor and the second with a dynamic one. The first one needs two threads, one must denote a partial interaction $static\_actor$ where the second one must denote a partial interaction $message_n$. The second rule needs, to represent the actor, both a partial interaction $behavior_n$ and a partial interaction $dynamic\_actor$. We first check that the actor's address is equal to the message's receiver and that the actor behavior label is equal to the message label. In the second rule, we also have to check the link between the behavior and the actor. We then define $v\_passing$ that describes the value passing due to both ζ operator and message handling.

4.1.4 Operational semantics

We now briefly describe how to use the preceding definitions to express in the non-standard syntax both an initial term and the computation of a transition according to a formal rule.

Initial configurations are obtained by launching a continuation in $init_s$ with an empty marker and an empty environment. We focus now on the interaction computation according to one of the two rules. First of all, we have to find the correct interaction, i.e. a set of two or three threads matching formal rule's synchronization constraints. Thus we can compute the interaction.

$$static\_trans_n = (2, components, compatibility, v\_passing) \quad \text{where:}$$

(i) $components = \{1 \mapsto static\_actor_n;\ 2 \mapsto message_n\}$
(ii) $compatibility = \{X_1^1 = X_1^2;\ X_2^1 = X_2^2\}$
(iii) $v\_passing = \{Y_1^1 \leftarrow X_1^1;\ Y_2^1 \leftarrow I^1\} \cup \left\{Y_{i+2}^1 \leftarrow X_{i+2}^2 \mid i \in [\![1; n]\!]\right\}$

$$dynamic\_trans_n = (3, components, compatibility, v\_passing) \quad \text{where:}$$

(i) $components = \{1 \mapsto behavior_n;\ 2 \mapsto dynamic\_actor;\ 3 \mapsto message_n\}$
(ii) $compatibility = \{X_1^2 = X_1^3;\ behavior\_set(I^1) = X_2^2;\ X_1^1 = X_2^3\}$
(iii) $v\_passing = \{Y_1^1 \leftarrow X_1^2;\ Y_2^1 \leftarrow X_2^2\} \cup \left\{Y_{i+2}^1 \leftarrow X_{i+2}^3 \mid i \in [\![1; n]\!]\right\}$

Figure 1. Non-standard semantics

We first remove interacting threads according to the type of their exhibited partial interactions; we choose a syntactic continuation for each thread; we compute dynamic data for each of these continuations. The dynamic data computation expresses both value passing, new restriction and new marker. The new threads are then put into the resulting configuration.

4.1.5 Correspondence

Proposition 4.1 (correspondence) *CAP and the non-standard semantics are in strong bisimulation.*

Proof Omitted. See [15] for a preliminary version.

4.2 Elements of abstract interpretation

Abstract interpretation [11] is a theory of discrete approximation of semantics. A fundamental aspect of this theory is that every semantics can be expressed as fixed points of monotonic operators on complete partial orders. A concrete semantics is defined by a tuple $(S, \subseteq, \bot, \cup, \top, \cap)$. Following [12], an abstract semantics is defined by a pre-order $(S^\#, \sqsubseteq)$, an abstract iteration basis $\bot^\#$, a concretization function $\gamma : S^\# \to S$ and an abstract semantics function $\mathbb{F}^\#$.

We approximate here the mobile systems' semantics as described in [14,23]. The collecting semantics of a configuration $\mathscr{C}_0$ is defined as the least fixed point of the complete join morphism $\mathbb{F}$:

$$\mathbb{F}(X) = (\{\epsilon\} \times \mathscr{C}_0) \cup \left\{ (u.\lambda, C') \middle| \exists C \in \mathscr{S}, (u, C) \in X \text{ and } C \xrightarrow{\lambda} C' \right\}$$

An abstraction $\left(\mathscr{C}^\#, \sqsubseteq^\#, \sqcup^\#, \bot^\#,\ \gamma^\#, C_0^\#, \rightsquigarrow, \nabla\right)$ in this framework must

define as usual a pre-order, a join operator, a bottom element, a widening operator (when abstract domains are infinite) as well as:

- the initial abstract configuration $C_0^\# \in \mathscr{C}^\#$ with $\{\epsilon\} \times \mathscr{C}_0 \subseteq \gamma(C_0^\#)$
- the abstract transition relation $\rightsquigarrow \in \wp(\mathscr{C}^\# \times \Sigma \times \mathscr{C}^\#)$ such that:
  $\forall C^\# \in \mathscr{C}^\#, \forall (u, C) \in \gamma(C^\#), \forall \lambda \in \Sigma, \forall C' \in \mathscr{C}$,

$$C \xrightarrow{\lambda} C' \implies \exists C'^\# \in \mathscr{C}^\#, (C^\# \overset{\lambda}{\rightsquigarrow} C'^\#) \text{ and } (u.\lambda, C') \in \gamma(C'^\#)$$

Such an abstract transition computes all the concrete transitions labeled λ from all possible C represented by $C^\#$.

The abstract counterpart $\mathbb{F}^\#$ of the $\mathbb{F}$ function is defined as:

$$\mathbb{F}^\#(C^\#) = \bigsqcup^\# \left(\left\{ C'^\# \mid \exists \lambda \in \Sigma, C^\# \rightsquigarrow^\lambda C'^\# \right\} \sqcup \{C_0^\#; C^\#\} \right)$$

4.3 Embedding type systems as abstract interpretations

The rationale of our embedding was to respect the semantics of our types.

First, our sub-typing rules, when closely inspected, express the natural flow of values occurring on communications. They also express the interplay between formal and real messages' parameters, as well as behavioral sub-typing, through contravariant sub-typing rules for behaviors (as communication is a disguised form of reduction, it naturally yields to contravariance on arguments). The control-flow is largely over-approximated, with the behavior typing rule, and this is a point where abstract interpretation could hardly do worse. As a consequence, we obviously need an abstract domain to represent this forward flow of values. But we do not need a very precise one to act as our type system, and in fact no history marker is necessary here.

Second, as for the linearity condition, things get a little more complex as sums of linear environments (as witnessed by the parallel and message typing rules) add a backward flow of information to the values' forward flow. This backward flow seems unavoidable and remains even if we manage to express this linearity property within our set-based constraint framework (which is possible but rather awkward). Indeed, in our abstract interpretation framework, the complexity of the linearity condition is not advocated by the abstract domain, which is equivalent to the usage modes, but rather by the computation of values when a communication occurs, all the more than our reference framework doesn't envisage such mixed information flows. The abstract transition relation must carry all these flows, and each transition must effectively compute a part of these flows until a fixed point is reached.

4.4 Abstract semantics

Properties expressed by a set of CAP non-standard configurations are approximated by a pair (c, l) where c denotes control-flow properties about configurations and l the linearity property about them.

4.4.1 Silly Control-flow abstract domain

The Silly Control-flow abstract domain is built upon the generic environment abstract domain defined by FERET in [14]. it allows to mimic the type system, which processes every branch of the analyzed term once and only once. Therefore, as we abstract here the collecting semantics in an operational way, we just have to drive the transition as would do the type system. The concretization function γ_{env} associates to each abstract element $(f_p)_{p\in\mathscr{L}_p} \in \mathscr{C}^{\#}_{env}$ the set of pairs $(u, C) \in \Sigma^* \times \mathscr{C}$ of reachable configurations with transitions u and such that each thread (q, id, E) in C satisfies $(id, E) \in \gamma_{I(q)} f_q$ where $I(q)$ denotes the interface of program point q.

Since we completely abstract markers, a thread environment is described by the set of values each variable can take (a subset of $\mathscr{L}_b$). An abstract element is a tuple of thread abstractions. For each interface V, we define the domain $Atom_V = (V \to \wp(\mathscr{L}_b))$. Each abstract element of such a domain is related to the set:

$$\gamma^C_V(f) = \{(id, E) | id \in \mathscr{M} \text{ and } E = \Pi_{v\in V}\{(l, id') | l \in f(v)\}\}$$

The structure $(\sqsubseteq^C_V, \sqcup^C_V, \bot^C_V)$ of the domain $Atom^C_V$ is defined component-wise with the usual set operators. The empty environment abstraction ϵ^C is defined as the map $[x \in V \mapsto \{\}]$. The abstract restriction $\nu^C(x, l, f)$ adds the name l to the set of x: $f[x \mapsto \{l\} \cup f(x)]$. The abstract garbage collection $GC^C(X, f)$ is defined as $f_{|V\cap X}$.

For any finite tuple $(V_i)_{1\le i\le n}$ of interfaces, we define $Molecule^C_{(V_i)}$ as the domain $\{(X, i) | 1 \le i \le n, X \in V_i\} \to \wp(\mathscr{L}_b)$. Each abstract element $f \in Molecule^C_{(V_i)}$ is related to the set $\gamma^C_{(V_i)}(f)$ of families $(id_i, E_i) \in \Pi_{1\le i\le n} Env^{\mathscr{M}}_{V_i}$ of pairs marker/environment such that $id_i \in \mathscr{M}$ and for any $v \in V_i, E_i(v) = (l, id)$ with $id \in \mathscr{M}$ and $l \in f(v, i)$. We define the abstract injection, which maps an atom to a molecule as the renaming: $INJ^C(f) = [(X, 1) \mapsto f(X)]$. The concatenation is a re-indexing:

$$(f_1 \bullet^C f_2)(X, i) = f_1(X, i) \text{ if } 1 \le i \le m \text{ and } f_2(X, i - m) \text{ if } m + 1 \le i \le n$$

The projection gives back an atom: $PROJ^C(f, k) = [v \in V_k \to f(v, k)]$. The extension removes any constraint about a set of variables: $NEW^C_\top(X, f) = f[v \in X \mapsto \mathscr{L}_b]$, and the synchronization consists in computing the meet of

Let $C^\# \in \mathscr{C}^\#_{env}$ be an abstract configuration, let $(n, components, compatibility, v\_passing)$ be a reduction rule, let $(p_k)_{1\leq k\leq n} \in \mathscr{L}_p$ be a tuple of program points and $(pi_k)_{1\leq k\leq n} = (s_k, (param_{k,l}), (bound_{k,l}), cont_k)$ a tuple of partial interactions. We define $mol \triangleq reagents^\#((p_k), (param_{k,l}), C^\#)$.

$C \xrightarrow{(p_k)_k}_\# \bigsqcup\{C; new\_threads\}$ where:

- (i) $\forall k \in [\![1; n]\!], pi_k \in interaction(p_k)$; (ii) $mol \neq \bot_{(I(p_k))_k}$
- $mol' = marker\_value^\#(s_1, (p_k), mol, (param_{k,l}), (bound_{k,l}), v\_passing)$
- $new\_threads = launch^\#((p_k), (cont_k), mol')$

Figure 2. Operational abstract semantics for control-flow analysis.

all associated sets in an equivalence class:

$$SYNC^C(S, (p_i), f)(x, k) = \begin{cases} \{\} \text{ if } \exists (y, l), f(y, l) = \bot \\ \cap \{(y, l) | (y, l) \in (x, k)_{=_S}\} \end{cases}$$

Finally the marker computation is totally abstracted: $FETCH^C(p^i, f) = f$.

Those two domains with their primitives represent an abstraction of the control-flow similar to the one of the type system, which only checks if the message labels and the addresses are the same. The abstract semantics is given in Fig 2. The primitive $reagents^\#$ computes a molecule with interacting atoms and adds synchronization constraints. The primitive $marker\_value^\#$ computes value passing and $launch^\#$ computes the set of new launched threads.

4.4.2 Linearity abstract domain

The second domain allows us to express linearity in the sense of the type system. Our semantics is defined by an abstract transition relation. In fact, as explained before, the current relation has two ways of data-flow, expressing both launching of new threads and update of interacting threads. The first one computes variables' values for new launched threads. But the use of a name in next instants may affect its current usage mode and constitutes a backward flow. Therefore the second part of our transition relation updates interacting threads' values.

We define our linearity abstract semantics as follows: an abstract element *Atom* is associated to each thread; when computing a transition we gather *atoms* to make a *Molecule*; we then compute new atoms' values using the *molecule* and according to the used transition relation. Finally we compute the second flow back and build the new values for interacting threads.

The use of a name is described by an element of the set $L = \{\bot^L, \bullet, \circ, \top^L\}$, to which we provide a complete lattice structure $(L, \sqcup^L, \sqsubset^L)$ where: $\bot^L \sqsubset^L \bullet$, $\bot^L \sqsubset^L \circ$, $\bullet \sqsubset^L \top^L$, $\circ \sqsubset^L \top^L$ and finally $\bullet$ and $\circ$ are not comparable.

Let $C^\# \in \mathscr{C}^\#_{lin}$ be an abstract configuration, let $(n, components, compatibility, v\_passing)$ be a reduction rule, let $(p_k)_{1\leq k\leq n} \in \mathscr{L}_p$ be a tuple of program points and $(pi_k)_{1\leq k\leq n} = (s_k, (param_{k,l}), (bound_{k,l}), cont_k)$ a tuple of partial interactions. Let $local_k$ be the pair $((param_{k,l}), (bound_{k,l}))$.

$C \xrightarrow{(p_k)_k}_\# \bigsqcup\{C; new\_threads; interacting\_threads\}$ where:

- $modes \triangleq count\_mode^{\mathrm{Lin}}((p_k), (local_k), v\_passing, (cont_k), C^\#)$
- $new\_threads = launch^{\mathrm{Lin}}((p_k), (cont_k), modes)$
- $interacting\_threads = update^{\mathrm{Lin}}((p_k), new\_threads, (local_k), v\_passing)$

Figure 3. Operational abstract semantics for linearity analysis.

Let V be the interface of a thread. We focus here on the use of names in a term to infer linearity. Therefore we are only interested in a subset of V. We keep variables describing an actor's name: in installation, or as message arguments. Let V' be the subset of such variables in V. We define the domain $Atom_V = V' \to L$. The lattice structure $(\sqsubseteq^L_V, \sqcup^L_V, \bot^L_V)$ of the domain $Atom^L_V$ is defined component-wise. We now describe the operational semantics of the abstract transition relation, which is threefold:

(i) We first gather interacting abstract threads and compute synchronization constraints among them, using the primitive $count\_mode^{\mathrm{Lin}}$.

Let $=_S$ be the equivalence relation defined by the compatibility constraints of the matching rule, by value-passing and by creations of new addresses. We associate to each equivalence class C of $=_S$ a mode $m_C = \sum_{(x,k)\in C} L_k(x)$, with $+$ defined as: $\bullet + \bullet = \top^L$ and $\forall x \neq \bullet, y. x + y = x \sqcup^L y$.

(ii) We then have to compute a forward flow defined by value passing and new threads launching. The primitive $launch^{\mathrm{Lin}}$ computes the set of new abstract threads on program points $\{p_i\}$ such that if p_i is:
- an actor bound on address a, then $L_{p_i}(a) \leftarrow L_{p_i}(a) \sqcup^L \bullet$;
- a message sent to address a, then $L_{p_i}(a) \leftarrow L_{p_i}(a) \sqcup^L \circ$;
- a behavior containing a free address a, then $L_{p_i}(a) \leftarrow L_{p_i}(a) + L_{p_i}(a)$;

The other $L_{p_i}(v)$ are left untouched.

(iii) Finally, we update the abstract elements related to interacting threads. We compute the inverse function of the value passing of the formal rule for the message parameters. The mode of the equivalence class of a formal parameter flows back to the corresponding real parameter. The primitive $update^{\mathrm{Lin}}$ computes such a backward flow.

In the end, when a fixed point is reached, we consider each equivalence class $=_S$ determined by each possible transition and containing a variable a mapped to α occurring in new threads as νa^α or $\zeta(a^\alpha, \_)$. The original CAP term is said to be linear only if each such class has mode $\bullet$.

5 Conclusion

We have shown how to express a particular type system into an abstract interpretation framework. Such a translation allows to express behavior passing without the strong restrictions that occurred when using type systems. Though not formally stated yet, it seems that our embedding closely simulates our previous type-based analysis, as until now we have not observed more false alarms. For instance, both analyses are able to deduce that example 2.2 is indeed linear.

We are now able to refine further this abstraction and for instance change the control-flow domain to a better one (*cf.* [13,15]) in order to cope with occurrence counting properties (of messages, actors, ...). Moreover, the abstract interpretation framework allows to design relational abstract domains[17,19]. These extensions are seldom present in type systems, for complexity reasons.

Yet, some properties such as detection of orphan messages seem to be still more easily expressed in type systems. This is partly due to the special treatment of names in type systems, for which the classic context-sensitivity problem tends to fade away and also to the ability to propagate information forward or backward along the transition relation, through hand-crafted sub-typing relations. But more importantly, the denotational flavor of type systems, which allows a rather easy detection of cycles in flows of values, does not seem to have a natural counterpart in terms of an abstract transition relation usually coupled with widening operators.

We now intend to generalize this encoding of a type system in terms of abstract domains, with in mind its application to our other type-based analyses, such as detection of orphan messages.

References

[1] Gul Agha. *Actors: A model of concurrent computation in distributed systems.* MIT Press, Cambridge, Mass., 1986.

[2] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[3] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 163–173, New York, NY, 1994.

[4] Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Proc. of the 3rd Asian Computing Science Conference on Advances in Computing Science*, volume 1345 of *LNCS*, 1997.

[5] Jean-Louis Colaco, Marc Pantel, and Patrick Sallé. A set-constraint-based analysis of actors. In *Proc. of FMOODS'97*. Chapman and Hall, 1997.

[6] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in Actors . In *Proc. of FMOODS'99*, volume 139. Kluwer, B.V., 1999.

[7] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. An actor dedicated process calculus. In *Proc. of the ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming*, 1996.

[8] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Analyse de linéarité par typage dans un calcul d'acteurs. In *Actes des Journées Francophones des Langages Applicatifs*, 1997.

[9] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. From set based to multiset based analysis: A practical approach . In *Proc. of CP'98*. Pise Univ., 1998.

[10] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Static analysis of behavior changes in Actor languages. In Jean-Paul Bahsoun, Takanobu Baba, Jean-Pierre Briot, and Akinori Yonezawa, editors, *Object-Oriented Parallel and Distributed Programming*, pages 53–72. Hermès Science, 8, quai du Marché-Neuf, 75004 Paris, France, 2000.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*. ACM Press, 1977.

[12] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[13] Jérôme Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming*, 63.1, 2005. special issue on pi-calculus, 2005.

[14] Jérôme Feret. *Analysis of Mobile Systems by Abstract Interpretation.* PhD thesis, École polytechnique, Paris, France, february 2005.

[15] Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Static Safety for an Actor Dedicated Process Calculus by Abstract Interpretation . Rapport de recherche todo, IRIT, décembre 2005.

[16] Carl Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proc. of the 3rd International Joint Conference on Artificial Intelligence*, 1973.

[17] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133 – 151, 1976.

[18] Naoki Kobayashi. A type system for lock-free processes. *Information and Computation*, 177(2):122–159, 2002.

[19] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[20] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[21] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *POPL*, pages 367–378, 1995.

[22] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Proc. of LICS'93*. IEEE Computer Society Press, 1993.

[23] Arnaud Venet. *Static Analysis of Dynamic Graph Strutures in Untyped Languages*. PhD thesis, École polytechnique, Paris, France, december 1998.