Type-Based Locality Analysis

Álvaro Moreira
Universidade de Caxias do Sul, Centro de Ciências Exatas e Tecnologia
Departamento de Informática, Caxias do Sul - RS, Brazil
afmoreir@ucs.tche.br

Paul Steckler
Rice University, Department of Computer Science, Houston - Texas, USA
steck@cs.rice.edu

Abstract

We present a functional distributed programming language and we give a polymorphic type system that detects references certain not to escape from one processor to another. Such information can be used by a compiler to replace assignment and dereferencing operations performed on such *local references* with specialized variants, which are less costly than the originals.
Keywords: Static Program Analysis, Type Systems, Distributed Languages.

1 Motivation

A key principle of programming language design is that only the users of a language feature should have to pay for it. In Matthews 'Distributed Poly/ML [7], all programs, including those that do not make use of distribution, may have to pay for the language's distributed features. Even when a reference refers to a local data, assignments and reads are required to invoke a coherency protocol, just in case the data is not local. The goal of the research presented in this paper is to eliminate this overhead when possible.

We propose a type-based program analysis for a functional distributed language, based on Poly/ML, to collect information about which mutable data is *local* to a single processor. Such analysis can be used in a Distributed Shared Memory system in order to reduce the cost of keeping distributed copies of mutable data consistent. Operations to access or modify *local mutable data* will usually comprise only a few machine instructions to manipulate data in the store, avoiding the need to call a coherency protocol.

The locality analysis we propose takes the form of a conservative extension of the Hindley-Milner polymorphic type discipline [8] where types of mutable data are labelled with locality information.

Our working language is a simplified version of a functional language with references and with primitives for creating distributed threads. We give a small step operational semantics for it and, for clarity of presentation, we split the soundness result of the type system with respect to the semantics in two parts: we prove *type* soundness, not taking into account locality properties and then we prove *locality* soundness: references that are considered local by our type system are certain not to escape according to the operational semantics.

2 The language and its operational semantics

The terms of the language are given by the following grammar.

$$M \quad ::= \quad x \mid c \mid \texttt{fn}\, x.M \mid MM' \mid (M, M') \mid \texttt{if}\ M\ \texttt{then}\ M'\ \texttt{else}\ M'' \mid \texttt{let}\ x = M\ \texttt{in}\ M'$$

where c ranges over the set of constants of the language, among which the following subset will be relevant for our analysis $\{\texttt{fork}, \texttt{rfork}_\rho, \texttt{ref}, \texttt{deref}, :=, r, r', r'', \ldots\}$. The constant `fork` is a *fork constant*, used to create new local threads; $\texttt{rfork}_\rho$ ranges over a finite set of constants, one for each processor ρ in a fixed network of processors. Each $\texttt{rfork}_\rho$ is used to create new threads on processor ρ; `ref` is used to create references, and `deref` and $:=$ are used to read and modify them, respectively. We write $M := N$ for the application of the assignment operator $:=$ to the pair (M, N). We use $r, r', \ldots$ for reference constants that denote store locations. Other constants are integer constants, `true`, `false`, `unit`, `succ`, `pred`, `zero?`, `fst` and `snd`, all with their usual meaning. The type rules for recursive functions are very similar to those for functions. For this reason, and in order to keep the presentation short we omit recursive functions from the language. A term M is a *program* if it has no free variables and no reference constants. Values are given by the following grammar:

$$V \quad ::= \quad x \mid c \mid \texttt{fn}\, x.M \mid (V, V').$$

One would naturally expect such a language to have channels for inter-thread synchronous communication. However, the locality analysis we propose should work for any mutable object and we believe that in a language with channels for thread communication, references are more likely to be local to a single processor. The payoff of detecting local references is then potentially greater than of detecting local channels. We then deliberately omit channels from our treatment and adopt references as our mutable objects of study.

We give a small-step operational semantics for the language in a style similar to the semantics of concurrent ML given in [2], [9], and [11]. We use evaluation contexts in order to reduce the number of rules in the operational semantics [3]. Evaluation contexts are terms with a "hole" in them, given by the grammar:

$$\begin{array}{rcl} E[\,] & ::= & [\,] \mid E[\,]\ M \mid V E[\,] \mid (E[\,], M) \mid (V, E[\,]) \mid \texttt{if}\ E[\,]\ \texttt{then}\ M\ \texttt{else}\ M' \mid \\ & & \texttt{let}\ x = E[\,]\ \texttt{in}\ M \end{array}$$

The notation $E[M]$ means that the hole in the context $E[\,]$ is filled by the term M. The holes are used to indicate the syntactic location of the next reduction step. A *redex* is an application of the form $V\ V'$, a conditional expression $\texttt{if}\ V\ \texttt{then}\ N\ \texttt{else}\ M$ or a let expression $\texttt{let}\ x = V\ \texttt{in}\ M$.

At each step of evaluation a thread of execution is represented by a *thread component* (ρ, M) where ρ identifies the processor where the thread runs and M is the term associated to the thread at that point of execution.

A *thread component map*, or *thread map* for short, is a finite, non-empty map from *thread identifiers* to thread components. We write p and q as typical thread identifiers. We write Π for a thread map. The thread map $\Pi[p : \rho, M]$ is the same as Π, except that the thread of execution identified by p is running on processor ρ and M is its associated term.

A *store* is a finite, possibly empty map from store locations to values. We write S as a metavariable for stores. The store $S[r \mapsto V]$ is the same as S, except that the store location r is mapped to the value V. The operational semantics is divided in two parts. The sequential part is shown in Figure 1 and the concurrent part in Figure 2.

3 The Type system

Types are given by the following grammar:

$$\tau \quad ::= \quad \texttt{bool} \mid \texttt{int} \mid \texttt{unit} \mid \alpha^\phi \mid \tau\ \texttt{ref}^{\,\phi} \mid \tau \xrightarrow{\phi} \tau' \mid \tau \times^\phi \tau'$$

$$S, E[(\texttt{fn}\, x.M)\ V] \xrightarrow[\text{seq}]{} S, E[M[V/x]] \qquad S, E[\,\texttt{if true then}\ M\ \texttt{else}\ N] \xrightarrow[\text{seq}]{} S, E[M]$$

$$S, E[\,\texttt{if false then}\ M\ \texttt{else}\ N] \xrightarrow[\text{seq}]{} S, E[N] \qquad S, E[\,\texttt{ref}\ \ V] \xrightarrow[\text{seq}]{} S[r \mapsto V], E[r] \quad r \notin Dom(S)$$

$$S, E[\,\texttt{deref}\ \ r] \xrightarrow[\text{seq}]{} S, E[V] \quad S(r) = V \qquad S, E[r := V] \xrightarrow[\text{seq}]{} S[r \mapsto V], E[\texttt{unit}] \quad r \in Dom(S)$$

$$S, E[\,\texttt{let}\ x = V\ \texttt{in}\ M\ \texttt{end}\,] \xrightarrow[\text{seq}]{} S, E[M[V/x]]$$

Figure 1: Sequential evaluation.

$$\textsf{Seq}\ \frac{S, E[M] \xrightarrow[\text{seq}]{} S', E[M']}{S, \Pi[p : \rho, E[M]] \xrightarrow[\text{con}]{} S', \Pi[p : \rho, E[M']]} \qquad \textsf{Fork}\ \frac{q \notin Dom(\Pi) \cup \{p\}}{S, \Pi[p : \rho, E[\,\texttt{fork}\ \ \texttt{fn}\, x.M]] \xrightarrow[\text{con}]{} S, \Pi[p : \rho, E[\texttt{unit}]][q : \rho, M[\texttt{unit}/x]]}$$

$$\textsf{Rfork}\ \frac{q \notin Dom(\Pi) \cup \{p\}}{S, \Pi[p : \rho, E[\,\texttt{rfork}_{\rho'}\ \ \texttt{fn}\, x.M]] \xrightarrow[\text{con}]{} S, \Pi[p : \rho, E[\texttt{unit}]][q : \rho', M[\texttt{unit}/x]]}$$

Figure 2: Thread map evaluation.

where α ranges over *type variables*. Some types include a *locality label* $\phi \in \{\ell, \epsilon, \kappa, \kappa', \ldots\}$. We call ℓ the *local label*, ϵ the *escape label*, and $\kappa, \kappa', \ldots$ are locality variables. The label of a type τ, if it has one, is given by $\mathbf{lab}\,(\tau)$. Our concern is with the locality properties of references, so clearly the locality labels belong on reference types. A reference that may escape from one processor to another will have a type with ϵ as its locality label; a reference that cannot escape can safely be assigned a type with locality label ℓ. As an example, suppose that the expression below runs on processor ρ:

$$\texttt{let}\ x = \texttt{ref}\ 0\ \ \texttt{in}\ \ \texttt{rfork}_{\rho'}\ \ \texttt{fn}\, w.\, \texttt{deref}\ x$$

Using $\texttt{rfork}_{\rho'}$, a new thread is created on a processor ρ' to evaluate the body of the function $\texttt{fn}\, w.\, \texttt{deref}\ x$. The reference x, created on ρ, *escapes* to processor ρ', so its reference type must be tagged with ϵ. In contrast, in the expression

$$\texttt{let}\ x = \texttt{ref}\ 0\ \ \texttt{in}\ \ \texttt{fork}\ \ \texttt{fn}\, w.\, \texttt{deref}\ x$$

the reference x is a *local reference* as it does not escape to a processor different from the one where it was created. Its reference type can therefore be tagged with ℓ.

Function types, pair types and type variables also carry a locality label. A value that escapes from one processor to another may contain reference subterms, causing those references to escape with the value. So the function in the first example above will have a type tagged with ϵ and the function in the second example can have a type tagged with ℓ. Type variables carry a locality label because they may be instantiated to labelled types.

3.1 Locality polymorphism

Analogous to the well-known fact that polymorphic references are unsound [14], references in our type system cannot be locality polymorphic. Suppose a thread p runs on processor ρ and has the following expression associated to it

$$\texttt{let}\ \ x = \texttt{ref}\ 3\ \ \texttt{in}\ \ (\texttt{fn}\, y.\, \texttt{deref}\ x)\ (\texttt{rfork}_{\rho'}\ \ \texttt{fn}\, w.\, x := 5).$$

The program above is non-deterministic: the result returned by thread p can be either 5 or 3. If we admit references polymorphic in their locality, the body of the let expression above could be typed in an environment where x has the type scheme $\forall\kappa.\texttt{int ref}^{\kappa}$. Then, the first occurrence of x can have the type $\texttt{int ref}^{\ell}$ and the occurrence in the *rforked* function has type $\texttt{int ref}^{\epsilon}$. Based on this type and locality information a compiler could translate the program to:

$$\texttt{let } x = \texttt{ref } 3 \texttt{ in } (\texttt{fn } y.\, \texttt{deref}^{\ell}\, x)\ (\texttt{rfork}_{\rho'}\ \texttt{fn } w.\, x := 5)$$

where $\texttt{deref}^{\ell}$ is the specialised local version of `deref` (which does not rely on a coherency protocol). Although we have not given a semantics for $\texttt{deref}^{\ell}$ we can argue informally that the program above no longer has a non-deterministic behaviour: even if the assignment is executed before the dereferencing, updating the copy of x in processor ρ' with 5, the operation $\texttt{deref}^{\ell}$ on ρ cannot retrieve this most recent value as it does not call a coherency protocol.

3.2 Constrained types

Consider the term `ref (ref 3)`. If the outer reference escapes, it may be dereferenced, causing the inner reference to escape. Therefore, possible types for such a term include $\texttt{int ref}^{\epsilon}\texttt{ref}^{\epsilon}$, $\texttt{int ref}^{\epsilon}\texttt{ref}^{\ell}$ and $\texttt{int ref}^{\ell}\texttt{ref}^{\ell}$. Explicitly we want to exclude the type $\texttt{int ref}^{\ell}\texttt{ref}^{\epsilon}$. Such a type would give *unsound* locality information. We also want to exclude the type $\texttt{int ref}^{\kappa}\texttt{ref}^{\epsilon}$. If the outer reference is labelled with ϵ, then we will expect that the type of the value this reference points to should be also labelled as escaping. We say that the type $\texttt{int ref}^{\kappa}\texttt{ref}^{\epsilon}$ provides *unresolved* information about locality.

The same ideas also apply to pairs. Suppose we have a pair like `(ref 3,ref 0)`. We have to exclude a type like $\texttt{int ref}^{\ell} \times^{\epsilon} \texttt{int ref}^{\ell}$, for example. Were we allowed to assign such a type, then the pair might escape, carrying with it references that are labelled with ℓ.

In order to exclude undesired types, we introduce types with *constraints* on their locality labels. A *constrained type* $\tau\backslash C$ is a pair consisting of a type τ and a set C of *constraints* on locality labels possibly appearing in τ, each of the form $\phi \leq \phi'$. We say that a constraint $\phi \leq \phi'$ is *unsound* iff $\phi = \epsilon$ and $\phi' = \ell$, and *unresolved* iff $\phi = \epsilon$ and ϕ' is a locality variable.

Definition 1 *A constraint set C is* valid *iff no constraint in its transitive closure is unsound or unresolved. A constraint set is* invalid *iff it is not valid.*

We will see that rules of the type system form, at their conclusions, the union of the constraint sets of their premises. It will be assumed that type derivations producing invalid constraint sets will be rejected.

Lemma 1 *If C and C' are valid constraint sets, then $C \cup C'$ is also valid.*

Proof. From the definition of valid constraint sets. ∎

3.3 The typing rules

Our constrained types can have locality variables as well as type variables, hence a *constrained type scheme* has the form $\forall\vec{\alpha}\vec{\kappa}.\,\tau\backslash C$, where $\vec{\alpha}$ and $\vec{\kappa}$ are sequences of type and locality variables, respectively. A *term type context* maps variables to constrained type schemes. A *store type context* maps store locations to constrained types.

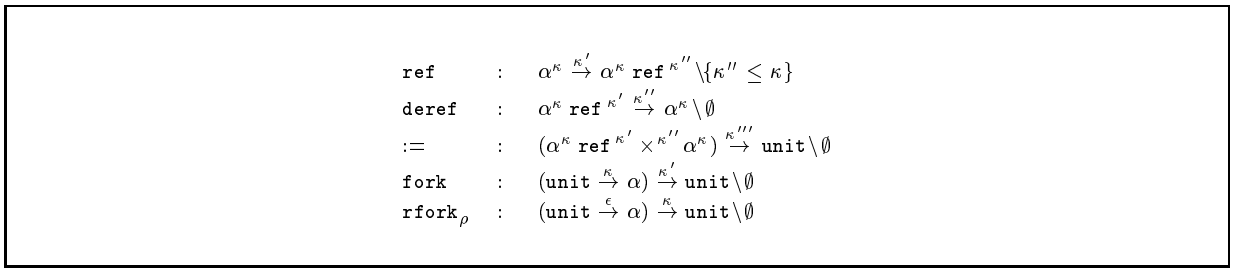

Figure 3: Constrained type schemes for constants.

Type judgements have the form $\Gamma; \Delta \vdash M : \tau \backslash C$ where M is a term and Γ and Δ are term and store type contexts respectively and C must be a valid constraint set. The typing rules also use the notions of generalisation of constrained types and instantiation of type schemes. Generalisation is similar to the usual definition, but also generalises locality variables:

Definition 2 (Generalisation) *$Gen(\tau \backslash C, \Gamma, \Delta) = \forall \vec{\alpha}\vec{\kappa}.\, \tau \backslash C$ where $\lceil \vec{\kappa} \rceil = FLV(\tau) \setminus FLV(\Gamma, \Delta)$ and $\lceil \vec{\alpha} \rceil = FTV(\tau) \setminus FTV(\Gamma, \Delta)$.*

Instantiation of constrained type schemes uses the usual notion of instantiation of regular type schemes, except that our substitutions Z consist of a pair (Z^{type}, Z^{loc}) containing a type substitution and a locality substitution.

Definition 3 (Instantiation) *We say that $\tau' \backslash C'$ is an* instance *of $\forall \vec{\alpha}\vec{\kappa}.\tau \backslash C$, written $\forall \vec{\alpha}\vec{\kappa}.\tau \backslash C \succ \tau' \backslash C'$ iff for some extended type substitution $Z = (Z^{type}, Z^{loc})$ such that $Dom(Z^{loc}) = \lceil \vec{\kappa} \rceil$ and $Dom(Z^{type}) = \lceil \vec{\alpha} \rceil$ we have $Z(\tau \backslash C) = \tau' \backslash C'$.*

Another component of our type system is the following predicate, whose role in the type system will be explained in the following subsection:

Definition 4 esc?(Γ, Δ,M) *holds iff for all x in $FV(M)$, $\Gamma(x)$ can be instantiated to a type with no locality label or to a type labeled with ϵ, and for all r in M, $\Delta(r)$ is a type labelled with ϵ.*

The rules of the type system appear in Figures 3 and 4. Figure 3 gives the constrained type schemes of all constants other than references. In the figure, all type and locality variables are implicitly universally-quantified. Each constant c has a constrained type scheme given by **typeof**(c).

3.4 Types and locality

Our type system detects references that may escape from one processor to another – the others, in a result that we prove in Section 5, are sure to be local. There are four mechanisms in the type system to enforce those cases where types should be labelled with ϵ.

The **first** mechanism is the type scheme for $\mathtt{rfork}_\rho$, as shown in Figure 3, which requires its argument to have a function type labelled with ϵ. This is sensible, since applying a remote fork constant in a parent thread spawns a child thread on a possibly different processor.

The **second** mechanism requires the predicate **esc?** to hold when typing functions with label ϵ or with a locality variable (rule ABS_ϵ in Figure 4). The predicate **esc?**$(\Gamma, \Delta$**,**$\mathtt{fn}\ x.M)$

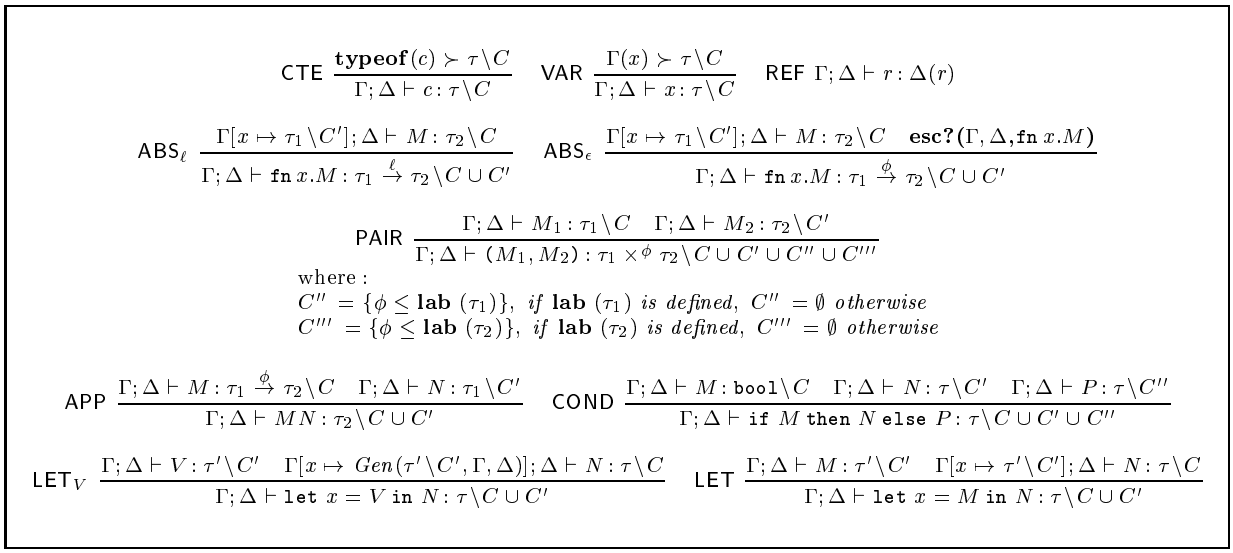

Figure 4: Typing rules.

only holds if $\mathtt{fn}\, x.M$ has no references labelled with ϵ and all its free variables can have valid escaping types. If a function is to be polymorphic in its locality it may have an escaping instance. For this reason the type rule used for deriving $\Gamma;\Delta \vdash \mathtt{fn}\, x.M : \tau_1 \xrightarrow{\kappa} \tau_2 \backslash C$ should also use the predicate $\mathbf{esc?}(\Gamma;\Delta,\mathtt{fn}\, x.M)$.

The **third** mechanism is a combination of the type scheme we assign to the constant `ref` in Figure 3 and the assumption that type derivations with invalid constraint sets are rejected. When `ref` is applied, if the resulting reference type is labelled with ϵ then the "inner" type cannot be labelled with ℓ or with a locality variable as that would produce an invalid constraint set.

The **fourth** mechanism is analogous to the previous one. It is a combination of the rule for typing pairs (PAIR) and the assumption that type derivations producing invalid constraint sets are rejected. So if the type of a pair (M_1, M_2) is labelled with ϵ then τ_1 and τ_2, the type of its components cannot be labelled with ℓ or with a locality variable.

4 Type soundness

In order to prove type soundness we follow the syntactic approach of Wright and Felleisen [16] and Harper [5] : we first prove type preservation, then we prove that expressions that may lead to execution errors are untypable.

In proofs for the Hindley-Milner polymorphic type discipline, it is essential to prove a *substitution lemma* which, in its turn, is used to prove type preservation ([14], [16]). In their proofs of Subject Reduction (Type Preservation), Wright and Felleisen are able to do with a single Substitution Lemma (Lemma 4.4 in [16]). In their β transition the type of the variable is the same as the type of the value to be substituted. In their rule for let expressions the type of the variable is a generalisation of the type of the value. For this reason only one lemma, assuming a quantified type for the variable, is necessary.

Our introduction of constrained types though requires two substitution lemmas. The crucial difference here is that the constraint sets in the constrained type of the variable and in the constrained type of the value are not known to be the same. A single lemma is then not possible: a quantified version of the type of the variable does not correspond to a generalisation of the value's type.

Lemma 2 (Polymorphic Substitution) *If* $\Gamma[x \mapsto Gen(\tau' \backslash C', \Gamma, \Delta)]; \Delta \vdash M : \tau \backslash C$, $x \notin Dom(\Gamma)$, *and if* $\Gamma; \Delta \vdash V : \tau' \backslash C'$ *then* $\Gamma; \Delta \vdash M[V/x] : \tau \backslash C$.

Proof. *By induction on the height of the typing derivation for* M. ∎

Unlike the Polymorphic Substitution lemma, our other substitution lemma assumes an unquantified constrained type scheme for the variable x to be substituted.

Lemma 3 (Ordinary Substitution) *If* $\Gamma[x \mapsto \tau' \backslash C']; \Delta \vdash M : \tau \backslash C$, $x \notin Dom(\Gamma)$ *and* $\Gamma; \Delta \vdash V : \tau' \backslash C''$ *then* $\Gamma; \Delta \vdash M[V/x] : \tau \backslash C'''$, *for some* $C''' \subseteq C \cup C''$.

Proof. By induction on the height of the typing derivation for M. ∎

The next definition states the requirements for a store typing context to be a typing for the references in the domain of a store.

Definition 5 *A store type context* Δ respects *a store* S *given a term context* Γ, *iff* $Dom(S) \subseteq Dom(\Delta)$, *and for all* $r \in Dom(S)$, *there exists* τ *such that* $\Gamma; \Delta \vdash S(r) : \tau \backslash C$ *for some constraint set* C, *and* $\Delta(r) = \tau\ \mathtt{ref}^{\ \phi} \backslash C'$ *where* C' *is valid and, if* **lab** (τ) *is defined, then* $\{\phi \leq \mathbf{lab}\ (\tau)\}$ *is in the transitive closure of* C'.

We write $\Gamma, \Delta \models S$ to say that Δ respects S given a term type context Γ. We say that Δ' *extends* Δ iff $Dom(\Delta') \supseteq Dom(\Delta)$ and for all $r \in Dom(\Delta)$, $\Delta'(r) = \Delta(r)$; we write $\Delta' \supseteq \Delta$ to indicate this relation. The next theorem says that types are preserved by sequential evaluation.

Theorem 1 (Sequential Type Preservation) *Suppose* $\Gamma, \Delta \models S$ *and* $\Gamma; \Delta \vdash M : \tau \backslash C$. *Suppose also that* $S, M \xrightarrow[\mathsf{seq}]{}{}^{*} S', M'$. *Then there exist* Δ' *and* C' *such that:* $\Delta' \supseteq \Delta$, $\Gamma, \Delta' \models S'$ *and* $\Gamma; \Delta' \vdash M' : \tau \backslash C'$.

Proof. By induction on the length of sequential evaluation. ∎

The type of terms in threads is preserved by concurrent evaluation. It also says that threads continue across evaluation, remaining on the same processor where they are created.

Theorem 2 (Concurrent Type Preservation) *Let* S, Π *be a configuration and* Δ *be such that* $\Gamma, \Delta \models S$ *for a term type context* Γ. *Let* p *be any thread identifier in* $Dom(\Pi)$ *and* $\Pi(p) = (\rho, M)$ *for some processor* ρ *and term* M. *Suppose that* $\Gamma; \Delta \vdash M : \tau \backslash C$. *Suppose also that* $S, \Pi \xrightarrow[\mathsf{con}]{}{}^{*} S', \Pi'$. *Then* $\Pi'(p) = (\rho, M')$ *for some term* M' *and there are* Δ' *and* C' *such that:* $\Delta' \supseteq \Delta$, $\Gamma, \Delta' \models S'$ *and* $\Gamma; \Delta' \vdash M' : \tau \backslash C'$.

Proof. By induction on the length of the reduction sequence. ∎

Type preservation is not enough for proving type soundness. The rules of a type system not well designed could assign a type for an expression with a faulty sub-expression, and this type would be preserved by the evaluation rules. We still have to prove that faulty threads, i.e, threads with faulty expressions cannot be typed.

If arbitrary configurations are allowed, then even faulty threads can be typed: the evaluation of p in $S, [p : \rho, (\mathtt{fn}\, x.\, x\ y)\ \mathtt{succ}\ ]$ leads to the configuration $S, [p : \rho, \mathtt{succ}\ \ y]$ where p is faulty. But one can easily show that $\Gamma; \Delta \vdash \mathtt{succ}\ \ y : \mathtt{int} \backslash C$ for any Δ and any Γ such that $\Gamma(y) = \mathtt{int} \backslash C$. Another example: the evaluation of p in $[r' \mapsto 3], [p : \rho, (\mathtt{fn} x.\ \mathtt{succ}\ \ (\mathtt{deref}\ x))\ r]$ leads to configuration $[r' \mapsto 3], [p : \rho, \mathtt{succ}\ \ (\mathtt{deref}\ r)]$ where p is faulty (in this case because $r \notin Dom([r' \mapsto 3])$. But again, $\Gamma; \Delta \vdash \mathtt{succ}\ \ (\mathtt{deref}\ r) : \mathtt{int} \backslash C$ for any Γ and Δ such that $\Delta(r) = \mathtt{int}\, \mathtt{ref}^{\ \phi} \backslash C$.

If a thread is faulty in a *well-formed configuration* then its associated term is untypable. We say a configuration S, Π is well-formed if none of the values in S and none of the terms in Π have free variables. Also, all reference constants which occur in values in the store or terms in Π, should be in the domain of S. This is not a restrictive assumption since all the computations of a program start in a well-formed configuration and well-formedness is preserved by evaluation.

Lemma 4 *If* S, Π *is well-formed and* $S, \Pi \xrightarrow[\mathsf{con}]{}{}^{*} S', \Pi'$ *then* S', Π' *is well-formed.*

Proof. *By induction on the number of evaluation steps.* ∎

Lemma 5 (Faulty Threads are Untypable) *If* p *is a faulty thread in a well-formed configuration* S, Π *and* $\Pi(p) = (\rho, M)$ *then* M *is untypable.*

Proof. If p is faulty, then M is not a value. Then M can be uniquely decomposed as $E[R]$ where R is a redex. We assume that there is a typing for $E[R]$ which implies that there is a typing for R, and we try to get a contradiction from this. Since S, Π is well-formed, p is faulty iff R is a redex of one of these forms:
- `deref` V, where V is not a reference constant,
- $V_1 := V_2$, where V_1 is not a reference constant,
- `fork` V, $\mathtt{rfork}_\rho\, V$ where V is not a function abstraction,
- `if` V `then` M `else` N where V is neither `true` nor `false`,
- $V_1\, V_2$ where V_1 is neither a function constant nor a function abstraction.

It is easy to show that if R has one of the forms above it is untypable. As a consequence $E[R]$ is also untypable. ∎

5 Locality soundness

We define three notions of *reachability* of local references that will be used to state locality soundness and we prove some of their properties.

The first notion is the set $\mathit{Reach}^{\Delta}(M)$ of reference constants of M which are local according to store type context Δ. We define this set as follows:

$$\mathit{Reach}^{\Delta}(M) \;=\; \{\, r \in \mathit{Refs}(M) \cap \mathit{Dom}(\Delta) \mid \mathbf{lab}(\Delta(r)) \neq \epsilon \,\}.$$

The following lemma says that if the type of a value is escaping then it has no local references as subterms.

Lemma 6 *If* $\Gamma; \Delta \vdash V : \tau \backslash C$ *and* $\mathbf{esc?}(\tau)$ *then* $\mathit{Reach}^{\Delta}(V) = \emptyset$.

Proof. By induction on the height of the type derivation for V and the definition of *Reach*. ∎

The set of reachable local references from a term M with respect to S and Δ is given by the following recursive definition:

$$\mathit{Reach}^{\Delta,S}(M) \;=\; \mathit{Reach}^{\Delta}(M) \;\cup \bigcup_{r \in \mathit{Refs}(M) \cap \mathit{Dom}(S)} \mathit{Reach}^{\Delta,S}(S(r)).$$

More formally, $\mathit{Reach}^{\Delta,S}$ is the least fixed point of the functional $F^{\Delta,S} : D \to D$ defined by

$$F^{\Delta,S}(f^{\Delta,S})(N) \;=\; \mathit{Reach}^{\Delta}(N) \;\cup \bigcup_{r \in \mathit{Refs}(N) \cap \mathit{Dom}(S)} f^{\Delta,S}(S(r))$$

where D is the set of functions from terms of the language to sets of references, ordered by $f^{\Delta,S} \sqsubseteq_D g^{\Delta,S}$ iff $f^{\Delta,S}(M) \subseteq g^{\Delta,S}(M)$ for all terms M. The least fixed point exists because D is clearly a directed complete partial order with a least element $\bot \stackrel{\text{def}}{=} \lambda M.\, \emptyset$, and one readily checks that $F^{\Delta,S}$ is continuous (Theorem 4.11, [4]).

The following lemma says that, under certain conditions, if the type of a value is escaping then no local references can be reached from it.

Lemma 7 *If* $\Gamma, \Delta \models S$ *and* $\Gamma; \Delta \vdash V : \tau \backslash C$ *and* $\mathbf{esc?}(\tau)$ *then* $Reach^{\Delta,S}(V) = \emptyset$.

Proof. We prove by fixed point induction [15] that $Reach^{\Delta,S}$ satisfies the property

$$P(f^{\Delta,S}) \stackrel{\text{def}}{=} \textit{for any value } V \textit{ if } \Gamma; \Delta \vdash V : \tau \backslash C \textit{ and } \mathbf{esc?}(\tau) \textit{ then } f^{\Delta,S}(V) = \emptyset.$$

For clarity, we omit the superscripts Δ and S as they are clear from the context. First we prove that P is admissible: one has to show that if each member of an ascending chain $f_0 \sqsubseteq f_1 \sqsubseteq f_2 \ldots$ satisfies the property, so does its *supremum* $\bigsqcup_{n\geq 0} f_n$. But this is immediate because $(\bigsqcup_{n\geq 0} f_n)(M) = \bigcup_{n\geq 0} f_n(M)$.

For the base case: we have to show that $P(\bot)$ holds which is also immediate.

Now the induction step: assume that $P(f)$ holds. We have to conclude that so does $P(F(f))$. In order to conclude this, we assume that both the conditions

$$\Gamma; \Delta \vdash V : \tau \backslash C \quad \textit{and} \quad \mathbf{esc?}(\tau) \tag{1}$$

hold, and from this we deduce that $F(f)(V) = \emptyset$.

By Lemma 6 and (1) above we have

$$Reach^{\Delta}(V) = \emptyset. \tag{2}$$

Hence, by (2) above and the definition of $Reach^{\Delta}$, if $r \in Refs(V)$ we have $\Delta(r) = \tau' \,\mathtt{ref}^{\,\epsilon} \backslash C'$ for some type τ' and some constraint set C'. Because $\Gamma, \Delta \models S$, if the same r is such that $r \in Dom(S)$, then

$$\Gamma; \Delta \vdash S(r) : \tau' \backslash C'', \tag{3}$$

for some constraint set C''. Also by the definition of respects (Definition 5) τ' is either ground or is labelled with ϵ, so we have

$$\mathbf{esc?}(\tau'). \tag{4}$$

Because $P(f)$ holds by hypothesis, by (3) and (4) we have

$$f(S(r)) = \emptyset, \textit{ for } r \in Refs(V) \cap Dom(S). \tag{5}$$

Therefore, $F(f)(V) = \emptyset$ since

$$\begin{aligned} F(f)(V) &= Reach^{\Delta}(V) \cup \bigcup_{r \in Refs(V) \cap Dom(S)} f(S(r)) \\ &= \emptyset \cup \bigcup_{r \in Refs(V) \cap Dom(S)} f(S(r)) \qquad \text{by (2)} \\ &= \emptyset \cup \emptyset \qquad \text{by (5)} \end{aligned}$$

which concludes the proof. ∎

We finally give our third notion of reachability. The set of local references reachable from processor ρ in thread map Π, given a store type context Δ and a store S is defined as follows

$$Reach^{\Pi}_{\Delta,S}(\rho) \;=\; \bigcup_{q \in Dom(\Pi) \wedge \Pi(q) = (\rho, M)} Reach^{\Delta,S}(M).$$

The following lemma says that, under certain conditions, if *the sets of local references reachable from different processors are disjoint* they remain so along evaluation. Before, we define the set of processors associated with a thread map Π as follows

$$ProcSet(\Pi) = \{\rho \mid \exists\, q \text{ such that } \Pi(q) = (\rho, M)\}.$$

Lemma 8 *Let S, Π be a configuration, and assume that all threads in Π are well-typed in a store type context Δ such that $\Gamma, \Delta \models S$. Suppose that $Reach^{\Pi}_{\Delta,S}(\rho) \cap Reach^{\Pi}_{\Delta,S}(\rho') = \emptyset$ for all distinct $\rho, \rho' \in ProcSet(\Pi)$. Suppose also that $S, \Pi \xrightarrow[\mathsf{con}]{}{}^{*} S', \Pi'$. Then there exists a Δ' that respect S' given Γ, $\Delta' \supseteq \Delta$, and, for all such Δ'*

$$Reach^{\Pi'}_{\Delta',S'}(\rho) \cap Reach^{\Pi'}_{\Delta',S'}(\rho') = \emptyset \text{ for all distinct } \rho, \rho' \in ProcSet(\Pi').$$

Proof. By induction on the length of the reduction sequence. ∎

An immediate consequence of the previous lemma is that if a local reference is reachable from just one processor in a configuration it remains so along evaluation, but that lemma does not say anything about whether the processor where the local reference is reachable from is the same.

Lemma 9 *Let S, Π be a configuration such that all threads in Π are well-typed under a store typing context Δ such that $\Gamma, \Delta \models S$. Suppose that $Reach^{\Pi}_{\Delta,S}(\rho) \cap Reach^{\Pi}_{\Delta,S}(\rho') = \emptyset$ for all distinct $\rho, \rho' \in ProcSet(\Pi)$. Suppose also that $S, \Pi \xrightarrow[\mathsf{con}]{}{}^{*} S', \Pi'$. Then for any $\rho \in ProcSet(\Pi)$, $\rho' \in ProcSet(\Pi')$, $r \in Reach^{\Pi}_{\Delta,S}(\rho)$ and $r \in Reach^{\Pi'}_{\Delta',S'}(\rho')$, where $\Delta' \supseteq \Delta$ respects S', implies $\rho = \rho'$.*

Proof. By induction on the length of the reduction sequence. ∎

Finally we can state and prove the main result. In the following, we write *subterm of a thread* meaning *subterm of the term associated to a thread.*

Theorem 3 (Locality Soundness) *Let P be a well-typed program, T be any sequence of configurations in evaluations starting with P. Suppose that the first occurrence of a reference r in T is in configuration S_i, Π_i as a subterm of a thread running on processor ρ. Suppose also that $\mathbf{lab}\,(\Delta_i(r)) \neq \epsilon$ where Δ_i is a store typing context such that $\Delta_i \models S_i$. If in a later configuration S_j, Π_j of T, the reference r appears as subterm of a thread running on ρ', then $\rho' = \rho$.*

Proof. All evaluations of P start with a configuration S_0, Π_0 where S_0 is the empty store and $\Pi_0 = [p_0 : (\rho_0, P)]$. So the fact

$$\text{all threads in } \Pi_0 \text{ are well-typed} \tag{6}$$

holds by assumption, and the fact

$$\text{for all distinct } \rho, \rho' \in Dom(\Pi_0), Reach^{\Pi_0}_{S_0,\Delta_0}(\rho) \cap Reach^{\Pi_0}_{S_0,\Delta_0}(\rho') = \emptyset \tag{7}$$

holds trivially for any type context Δ_0.

By the Concurrent Type Preservation theorem, there exists Δ_i respecting S_i such that $\Delta_i \supseteq \Delta_0$. Hence, by Lemma 8 with (6) and (7) above we get that

$$\textit{for all distinct } \rho, \rho' \in Dom(\Pi_i), Reach^{\Pi_i}_{\Delta_i,S_i}(\rho) \cap Reach^{\Pi_i}_{\Delta_i,S_i}(\rho') = \emptyset. \tag{8}$$

By the Concurrent Type Preservation theorem there is Δ_j respecting S_j, $\Delta_j \supseteq \Delta_i$. Hence by (8) above and Lemma 9 we have that if $r \in Reach^{\Pi_i}_{\Delta_i,S_i}(\rho)$ and $r \in Reach^{\Pi_j}_{\Delta_j,S_j}(\rho')$ then $\rho = \rho'$. ∎

5.1 Related work

Locality issues in Distributed Poly/ML are addressed in [11] where a global dataflow analysis to detect local *channels* is presented. The syntactic approach to dataflow analysis used there built upon earlier work for sequential languages, for example, [12] and [13]. Such global analyses, contrary to the type-based analysis designed here, are inherently non-compositional.

That work uses a language that is similar to ours and it does a reachability analysis that detects some cases of mutable objects that go to processors different where they were declared but are not used in these processors. Such cases are not detected by our type-based analysis.

In [10] and [17] type-based approaches using subtypes are used to detect global and local capabilities of channels in distributed calculi based on Milner's π-calculus.

All the analysis cited above were proved to be correct in the sense that they do not infer that an object is local when in fact it escapes. In order to make a precise comparison between the amount of locality these analysis and ours detect it is necessary to develop a common framework in which to represent them.

For reasons of space, we omit a sound type inference algorithm for our type system which is presented in [1]. The algorithm always stop but does not necessarily yield a maximum number of local references. Not taking into account locality labels, it behaves exactly like $\mathcal{W}$ [8]. This fact provides evidence that the algorithm is complete.

6 Conclusions

We believe that our type-based approach can be useful for other distributed languages with mutable objects based on a shared memory system. Note that our type system requires that a program itself must contain information of which threads are to be executed remotely. Because of that our locality analysis cannot be readily used for parallel languages where the compiler decides which parts of a program will constitute concurrent threads. However, a compiler might make parallelisation and distribution decisions first, then apply the locality analysis.

Any non-specialised operation to access mutable data can be used wherever a specialised version, not calling a coherency protocol is expected. Because of that subtypes might seem to be a natural choice for a type-based locality analysis to detect locality. Our intention was to treat locality variables in a way analogous to the way the SML type discipline treats type variables. For this reason and because the type system of languages based on the SML type discipline does not support subtypes we decided to adopt variables ranging over locality labels.

References

[1] Álvaro F. Moreira. A type-based locality analysis for a functional distributed language. Technical Report ECS-LFCS-00-4186, LFCS, University of Edinburgh, April 2000.

[2] David Berry, Robin Milner, and David N.Turner. A semantics for ML concurrency primitives. In *Conference Record of 19st ACM Symposium on Principles of Programming Languages*, pages 84–97, January 1992.

[3] Matthias Felleisen, Daniel Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–37, 1987.

[4] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, Cambridge, MA, 1992.

[5] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, August 1994. See also note [6].

[6] Robert Harper. A note on: "A simplified account of polymorphic references" [Inform. Process. Lett. **51** (1994), no. 4, 201–206; MR 95f:68142]. *Information Processing Letters*, 57(1):15–16, January 1996. See [5].

[7] David C.J. Matthews. A distributed concurrent implementation of Standard ML. LFCS Report Series ECS-LFCS-91-174, University of Edinburgh, August 1991.

[8] Robin Milner. A theory of type polymorphism in programming. *Jornal of Computer Systems Sciences*, 17:348–75, 1978.

[9] J.H. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, 1992.

[10] Peter Sewell. Global/local subtyping and capability inference for a distributed pi-calculus. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *25th Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *LNCS*, pages 695–706. Springer, July 1998.

[11] Paul Steckler. Detecting local channels in Distributed Poly/ML. Technical Report ECS-LFCS-96-340, Laboratory for the Foundations of Computer Science, Edinburgh University, January 1996.

[12] Paul Steckler and Mitchell Wand. Selective thunkification. In Baudouin Le Charlier, editor, *Proceedings of the 1st International Static Analysis Symposium*, volume 864 of *LNCS*, pages 162–78, Berlin, Heidelberg, and New York, 1994. Springer-Verlag.

[13] Paul Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–46, January 1997.

[14] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(2):1–34, 1990.

[15] Glynn Winskel. *The Formal Semantics of Programming Languages : an Introduction.* MIT Press, Cambridge, MA, 1993.

[16] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1992.

[17] Nobuko Yoshida and Matthew Henessy. Subtyping and locality in distributed higher order mobile processes. In *CONCUR'99*, LNCS. Springer, 1999.