A Formal Language for Programming with Distributed Resources and Scopes*

WANG Xu, HUANG Tao, FENG Yu-lin

(Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080, China)
E-mail: {wx,tao}@otcaix.iscas.ac.cn
Received March 31, 2000; accepted November 3, 2000

Abstract: This paper presents a small language for distributed computation with agent mobility—— Scope language. The language is different from most existing work on distributed and mobile computation, which usually take some variants of π-calculus as their basis. The core of Scope language is a specially designed λ-like calculus with resources. It enables Scope language to directly model memory like resources, instead of indirectly using process/channel as in π-calculus. Furthermore, Scope language gives a novel treatment to the notion of location, which is called Scope here. Scope and memory-like resources combined make Scope language complementary to most other work, and provide an alternative approach to modeling distributed and mobile systems, which feature the simplicity of implementation and the affinity with the programming model inherent in realistic language such as Obliq and Telescript.

Key words: mobile agent; resource; location; scope; formal language

1 Motivations

In recent years, a variety of calculi have been proposed as models of distributed and mobile computation. Among them, the most widely known are Ambient[1], Join[2], $D\pi$[3], Seal[4], π_I-calculus[5]. Most of these work are extensions of some existing calculi which provide new primitives for explicit modeling of resources distribution and agents mobility. Currently, variants of π-calculus are the most popular basis for such extension. As a result, in most cases the resources and agents of interest are actually channels and processes in process calculi.

Parallel to these theoretical works, research on programming languages for distributed and mobile systems are also being pursued actively. There emerged a large number of experimental programming languages, such as Obliq[6], Aglet, Odyssey, etc. In these languages, however, channels are not the major resources used in programming (at least not the only one). Instead, data storing memory cells are the resources more widely used. Unlike channels, whose use is for process communications only, the functionality of memory cells in programming is far more versatile. Though it is true that process/channel may be used to emulate memory cells, the emulation is arguably awkward and has lost the intrinsic simplicity of memory cells.

---

* This project is supported by the National 973 Fundamental Research Program of China under Grant No. G1998030404 (国家重点基础研究发展规划项目) and the National Natural Science Foundation of China under Grant No. 69833030 (国家自然科学基金). **WANG Xu** was born in 1971. He received his Ph. D. degree in Computer Science from Institute of Software, the Chinese Academy of Sciences in 2000. His research interests are software engineering, software agent and distributed object. **HUANG Tao** was born in 1965. He is a professor and doctoral supervisor at the Institute of Software, the Chinese Academy of Sciences. His research interests are software engineering, object technology, distributed computing and programming language/environment. **FENG Yu-lin** was born in 1942. He is a professor and doctoral supervisor at the Institute of Software, the Chinese Academy of Sciences. His research interests are software engineering, network distributed computing and theory of component-based software.

Hence, for a good modeling of mobile systems, we believe it is reasonable that memory cell be treated as basic construct in the formalism, and a new formalism that synergistically combine passive data with location and active agent is needed to help bridge the gap between theory and practice.

2 Background and Related Work

As stated in Ref. [3], the paradigm of mobile computation is centered around the notions of agent, resource and location. Agents are the active entity in the system, they access resources and interact with other agents to perform computation. An agent at one time resides at one location only; over time it may move from one location to another. Resources, on the other hand, are passive entities in the system; they do not perform computation and generally are fixed to a location since their creation time.

In modeling such a system, different calculi focus on different aspects of the system (Seal on security, Ambient on mobility, $D\pi$ on resource access control). So, the primitives they choose and the technical approach they adopt are all different. The differences, however, never hinder them from providing inspirations for our work; especially, the following works have influenced us in a significant way: Ambient[1], Seal[4] and $D\pi$[3].

· The core of the Ambient calculus is a minimal calculus for mobility, which only supports the notion of process (i.e. an agent) and ambient (i.e. a location plus a collection of agents located in it). Ambients are organized into a hierarchy, where agents perform computation via the mobility of ambient. An ambient moves with all its sub-ambients, and the movement is either up the hierarchy by migrating out of its parent or down the hierarchy by migrating into one of its siblings.

· The Seal calculus is a miniature language designed as a model of distributed and mobile programming. The basic notions it supports are very similar to those of Ambient, except that ambient is called seal here, and that it supports channels as resources. Seals are also organized into a hierarchy; but here agents perform computation by communicating along channels with each other. Like π-calculus, the communication is usually passing of names; but in other cases Seal calculus also supports passing of seals as means of seal migration. Channels are located as well as agents. An agent can only access resources located in the local seal or in one of its neighbouring seals.

· The $D\pi$ calculus is an extension of π-calculus to support the notion of location. So, the agents and resources are respectively processes and channels. Locations are not organized; they simply form a flat set. Agents and resources are located. An agent can communicate names only with its co-located agents (over local channels), whereas it can migrate from one location to any other location in the flat set.

Different calculi strive for different goals in their calculi designs. Ambient calculus is intended as an abstract model of mobile computation, and tries to reduce the primitives and notions in the formalism to a minima to facilitate the theoretical study of properties inherent in mobile computation. Although Cardelli's final goal is a programming language for Internet (or wide-area network), Ambient calculus considers no implementation issues and is still far away from a realistic programming language. This is also the case for $D\pi$, which follows the tradition of π-calculus extensions to distributed scenario as in the line of works of $\pi_{1/}$ calculus. On the other hand, Seal calculus is more pragmatic in this respect. Rather than striving for minimality, it tries to express key features of Internet programming directly, and there is already an implemented language based on it, JavaSeal.

The aim of this paper is to devise a formal language supporting distributed programming with agent mobility, whose design goals are the simplicity of implementation and the affinity with the programming model used in realistic distributed languages such as Obliq, Telescript, etc.

3 General Approach and Basic Concepts

Our approach of language design is different from most other works. In their approaches, a calculus for

concurrency, especially π-calculus, is taken as basis; thus a full set of communication and concurrency primitives is within the formalism. In our approach, a concurrent λ-calculus-like language with resources, SL language, is taken as basis, which is specially formulated by us to meet the designed goal of Scope language. It supports only a weak form of concurrency and provides no communication primitives; but memory-like resource is included in the formalism as a basic construct.

Although it would not be difficult to extend SL language with communication primitives and channels, for this paper we ignore this possibility, in order that the language is kept simple and the attention more focused.

Based on the small language, Scope language is formulated that supports the notion of agent, resource, and scope (i.e. location). Agents and resources are located in scopes, which in turn are organized roughly into a hierarchy, where agents move around to perform computation.

The agent in our language models very well the concept of thread in common programming languages like C and Java: it can fork to create new thread (agent), and can issue instruction to access variables (memory cell resources). Also, as two threads can not be combined to form a larger thread, our language does not support process composition operator as in process algebra-based formalism.

Resources here are in two forms: data resource that corresponds to data variable such as integer or char variable in C, and code resource that corresponds to function variables. The instruction, accordingly, has three types: fetch and assignment for accessing data resource, and invocation for accessing code resources.

Scope is the central concept underlying Scope language. Like ambient and seal, it is a variant of the notion of location. As it is an elaborated concept that will not be used until Section 5, we postpone its explanation to Section 5, where Scope language is formulated.

Our language is different from π-calculus significantly. But it has borrowed and extended one important notion of π-calculus, the name. In π-calculus, name is used to denote port, i.e. the capability to access a certain channel; whereas the channel itself, which is only an empty place for communication, need not be explicitly represented in the formalism. In Scope language, however, memory cell is a place for storing data; its explicit representation becomes mandatory in our language.

Consequently, name in Scope language appears both in construct denoting memory cell and in construct denoting capability to access it. We call the former a lock while the latter a key; a key can open a lock which has the same name as its. Keys can also be passed into function or stored as value in a lock (just like ports can be communicated as values).

In the rest of the paper, we will present the SL language first. Then, based on it, Scope language is formulated and the notion of scope is explained. Finally, we discuss and summarize Scope language and comment on future work.

4 SL Language

In general terms, the core of SL language is like a concurrent name passing λ calculus with resources (It is partially inspired by Boudol's work in Ref. [8]), and it is imperative and based on implicit continuation passing. In this section, we first introduce the syntax of the language in its entirety. Then, an informal explanation of the various constructs in the formalism is given. Finally, we summarize its operational semantics in terms of a reduction system.

4.1 Syntax

The syntax of the SL language is defined in the following table. The main syntax categories are Expression, Function, Continuation, Process, Resource and Environment.

Syntax of SL language	
$E::=x\|x;(y)\|x?\ (y).E\|x!\ (y).E\|$ fork $E'.E\|x(y)F$	Expression
$F::=(\lambda x)E$	Function
$C::=\mathscr{C}\|FC$	Continuation
$P::=0\|EC$	Process
$R::=x\langle y\rangle\|x\langle F\rangle$	Resource
$S::=R,T,\ldots\vdash P,Q,\ldots$	Environment

Like in π-calculus, the distinction between name and variable is removed; they are all names. The symbols x, $y,z,\ldots$ is used to range over an infinite set $\mathscr{N}$ of names. The binders are the x in λx and the y in $x?\ (y)$.

In addition, we assume in the rest of the paper the following abbreviations. $(\lambda)E$ is an abbreviation of $(\lambda x)E$ where x does not occur free in E; $u;()$ is an abbreviation of $u;(v)$ where u points to a code resource containing $(\lambda)E$; $u\langle\rangle$ is an abbreviation of a data resource whose contents may be arbitrary. $x!\ (y)$ and $x?\ (y)$ are abbreviations of $x!\ (y).z$ and $x?\ (y).z$ respectively, where we do not care about the final return value in z.

4.2 Explanation

Environment is the term in SL language, on which the reduction rules is defined. Other constructs of the formalism are just components used to build an environment. In many senses, environment is akin to the notion of configuration in Actor formalism[9]. Below we explain its constructs one by one.

1. Key and lock

   Key is denoted simply by a name in SL language; any free occurrence of name in an environment is a key. Keys are the capabilities to access resources. It is volatile data in the sense that it is communicable, duplicable, and storable.

   Locks are denoted by the construct $x\langle\rangle$. It denotes a lock that can be open only with key x. Locks are the memory cells that store "substance" like data or code. It is "persistent" in an environment and appears on its left hand side.

2. Resource, Expression and Function

   As discussed in previous section, resources are either data resources or code resources. A data resource, $x\langle y\rangle$, is a key y in a lock $x\langle\rangle$. A code resource, $x\langle F\rangle$, is a *function* F in a lock $x\langle\rangle$.

   Although instruction is an important notion in our language, we don't have a separate syntax category for it. Instruction is defined indirectly in the syntax category of expression. A fetch instruction, $x?(y)$, fetches a key, presumably z, from a lock named x and makes the substitution $[z/y]$ to the rest of the expression (see rule Red Fetch). An assignment instruction, $x!\ (y)$, puts key y into a lock named x (see rule Red Asgn). An invocation instruction, $x;(y)$, invokes a function, presumably F, from a lock named x, and is ready to call the function with a key y as argument (see rule Red Inv). A fork instruction, forkE', creates a new process P in the environment, and then P runs in parallel with the parent process (see rule Red Fork).

   An expression is something that can evaluate a value (i.e. a key). An expression may take several forms. It can be another expression prefixed with an instruction, which is either a fetch, an assignment, or a fork. Or, it can be simply an invocation instruction, which evaluates a function call (see rule Red Inv) that in turn reduces to another expression (see rule Red Call). In addition, an invocation instruction followed with a function is also a form of expression. The function here is used to contain the "statements" following the invocation instruction, which will accept the return value of the invocation. Before its evaluation, this form of expression will need a special structural transformation (see rule Struct ICP) to make rule Red Inv applicable.

A function is a "small" abstraction[10] of expression. It is monadic, but extending it to polyadicity is not difficult. The polyadic function will be able to take multiple arguments in a single call.

3. Process and Continuation

A process is either an inactivity (see rule Struct GC), 0, or an activity that first evaluates an expression and then uses the evaluated value to continue the "rest" of its activity, EC. The rest of the activity is called a continuation. A continuation is either a function to be continued with another continuation, FC, or simply a to-be-ended activity (see rule Red End), $\varnothing$.

4. Environment

An environment, $\vdash$, is a place where computation happens. On the left hand side of $\vdash$ is a multiset of resources; on its right hand side is a multiset of processes. Processes run in parallel to perform computation. Keys used in computation generally have matching locks on the left hand side of the environment.

4.3 Reduction semantics

Computation happens with the step-by-step reduction of an environment, where each step is prescribed by a reduction rule. The collection of reduction rules constitutes the operational semantics of SL language.

To make the reduction system simple, environments are identified up to the renaming of bound names as well as to the following structure congruence rules.

Structure Congruence	
$\vdash (x:(y)F)C \equiv \vdash x:(y)(FC)$	(Struct ICP)
$\vdash 0 \equiv \vdash$	(Struct GC)

Then, the rules of the reduction system are given below,

Reduction	
$x\langle F\rangle \vdash x:(y)C \rightarrow x\langle F\ \rangle \vdash (yF)C$	(Red Inv)
$x\langle z\rangle \vdash x?\ (y).EC \rightarrow x\langle z\ \rangle \vdash E[z/y]C$	(Red Fetch)
$x\langle z\rangle \vdash x!\ (y).EC \rightarrow x\langle y\ \rangle \vdash EC$	(Red Asgn)
$\vdash \mathrm{fork}E'.EC \rightarrow \vdash E'\varnothing, EC$	(Red Fork)
$\vdash y((\lambda x)E)C \rightarrow \vdash E[y/x]C$	(Red Call)
$\vdash x\varnothing \rightarrow \vdash 0$	(Red End)

All the rules above are given in the chemical abstract machine style[16].

5 The Scope Language

In this section, we first introduce the concept of scope. The syntax of Scope language is given in its entirety, and various constructs in the formalism are explained informally. After that, we give its operational semantics in terms of a labelled transition system, and specifics of the transition system are clarified. Lastly, the uses of the language are illustrated by examples.

5.1 Scope

Scope language is an extension of SL language. The central notion underlying the extension is that of scope, which is a variant of the notion of location.

In its simplest sense, a location is just a centre of activity[5], or a bounded place where computation happens[1]. This sense is the root for many variants of the notion, and it is embodied in our concept of environment in SL language.

After this common start, however, variants of the notion turn divergent on many an aspect: the organization of locations (a hierarchy or a flat set), the interaction between entities (local within one scope, or remote across

multiple scopes), or the form of mobility (mobility of agent or mobility of location).

For the aspect of location organization, our language resembles Ambient and Seal, where a location (i.e. an ambient or seal) is a named place consisting of a collection of processes. As locations are identified to processes in these calculi(i.e. they share the same syntactic category), some processes in a location are indeed locations. One location thus is really a composition of hierarchy of sub-locations.

In Scope language, the named place is an environment, which is called scope here and consists of a collection of processes and resources. But scope is not identified as either process or resource. Rather, we understand scope to be "substance", which can be stored in a lock and can form a resource, i.e. scope resource.

Therefore, scopes are not directly composable here. It is only after being locked in as resources, can they be used in composing other scopes. One scope, accordingly, has two names associated with it: one is the scope name (the name of the place); the other is the resource name (the name of the lock that encapsulates it).

Scope as substance is an interesting research topic, involving problems such as the storage of the substance and the movement of the substance(i.e. scope mobility). This paper will not tackle this topic, for we should concentrate on two other significant features of scope:

- A scope is a boundary, which delimits what entities (resources or agents) are inside it and what are outside.
- A scope is a reference frame, that is, an addressing system for locating resources inside its boundary.

In a hierarchy, the entities local at one scope or at any of its sub-scopes recursively, are all inside the scope's boundary. The boundary provides protection for these resources. To access them from outside the boundary, you need a key for the lock encapsulating the scope. Resources may simultaneously be inside a series of boundaries, each being the outer boundary of the next in the series. So there are several layers of protection for them. If we are to locate and access a resource in the hierarchy, we will need at least two things: first, a scope (one of the resource's outer boundary scopes) that is chosen as the reference frame; second, a cluster of keys for all the locks along the path from the reference frame down to the resource in the hierarchy.

When an agent is to access resources, usually one of its outer boundary scopes is specified (using scope name) as reference frame. If none is specified, the default reference frame is the agent's local scope. In general, a reference frame should be a common outer boundary for both the agent and the resource.

With this addressing system in place, an agent, if having necessary keys, can access resources located at anywhere in the hierarchy (In this aspect, it resembles Join calculus). Therefore, the interactions in our language are mostly remote. When it is a data resource, the agent can retrieve data directly from the remote scope of the resource. When it is a code resource, the agent will exert its mobility (i.e. agent mobility). It first migrates across the hierarchy to the remote scope of the resource, where it retrieves the function in it and starts its execution there. Then, after the function is finished, it returns to the original scope, resuming its suspended computation there (i.e. the continuation).

Based on this hierarchical model, it is possible to formulate a useful language for distributed systems as is done in other works. However, the strict hierarchical model has some problems with it. The most important one of them is that the flexibility of the addressing system compromises the security of distributed systems.

On that account, this paper has not directly adopted this model as it is. We have enhanced it with both some extensions and some restrictions in order to facilitate good programming practices that guarantee the security of distributed systems. The resulting model is our Scope language.

The extensions to the model are two folds.

Introspection: In a strict hierarchy, the reference frames that an agent may take need to be one of its outer boundary scope. Nevertheless, by introspection an agent may take additional scopes as reference frame as well.

Public modifier: In a strict hierarchy, to access resources in a sub-scope, an agent must use a cluster of keys

to penetrate the sub-scope. But with a new modifier public for resources, resources at a sub-scope may sift out and be directly accessable in the local scope using a single key.

A scope may statically introspect another scope, so that any agent inside the former's boundary will be able to take the latter as its reference frame. The reference frames that an agent may take due to this static introspection and those due to being its outer boundary are collectively called its static reference frames.

An agent may also dynamically introspect a scope, so that the latter can be used directly as reference frame by the former. They are called its dynamic reference frames.

Usually, resources are directly accessable only in the local scope. Public resources, however, can sift through this local boundary and be directly accessable in the parent scope as well. This sifting is transitive if the local scope is a public resource in the parent, too.

In addition to these extensions, we have also made three restrictions.

• An agent may dynamically introspect a scope only if the latter is a resource directly accessable in one of its static reference frame.

• A scope may only statically introspect what its co-located agent (that is, the scope resource and the agent are co-located) may dynamically introspect.

• Within any of its reference frame (dynamic or static), an agent may only access the resources that is directly accessable.

The restrictions provide a kind of protection for sub-scopes inside a reference frame. Programmers are not allowed to open sub-scopes and to dig arbitrarily deep into the hierarchy.

5.2 Syntax

In Scope language, we keep the distinction between name and variable. Let $\mathscr{X}$ be an infinite set of variables, and $\mathscr{N}$, $\mathscr{L}$ be non-overlapping infinite sets of names. We will have meta-variables, $m,n\in\mathscr{N}$, $l,s\in\mathscr{L}$, and $x,y,z\in\mathscr{X}$, where m,n are resource names used in key/locks and l,s are scope names denoting reference frames.

Syntax of Scope language

$v,u::=i\mid I::i\mid v.i$	Value
$k::=x\mid v\mid x.i$	Key
$r::=k\mid v::i\mid x::i$	& Cluster (of keys)
$E::=k\mid r:(k)\mid r?\ (x).E\mid r!\ (k).E\mid$ fork $E.E\mid r:(k)F$	Expression
$F::=(\lambda x)E$ (where variables are all bound)	Function
$M::=F\mid I::F\mid V::F$	Return
$C::=\varnothing\mid MC$	Continuation
$P,Q::=0\mid EC$	Process
$H::=\rightarrow v,s\rightarrow u,\ldots$	Introspections
$R,T::=n\langle v\rangle\mid n\langle F\rangle\mid n\langle S(H)\rangle$	Resource
$S(H)::=\epsilon R,\epsilon T,\ldots\vdash\sharp P,Q,\ldots$	Scope

Here, $i::=n\mid i.i$, $I::=i\mid I::I\mid I.I$, $V::=I\mid I::I\mid V::I\mid V.I$, $\epsilon::=\uparrow\mid\downarrow$, and the binders are the x in λx and $r?\ (x)$.

5.3 Explanation

Scope language has 11 major syntax categories, with scope as the term in the language. Below these syntax categories are explained one by one.

1. Key, Value and Cluster

Although the construct for locks remains the same in the new language, i.e. $n\langle\rangle$, the construct for keys, k, is extended substantially.

There are local keys, i, that is, keys with no reference frame specified and thus using the local scope as default. Local keys may be denoted by a single name, e.g. m, which is a simple key to a resource $m(X)$ at the local scope. Local keys may also be denoted by a sequence of names connected with '.', e.g. $n.m$, which is a compound key to a public resource $\uparrow m(X)$ that sifts out from the sub-scope encapsulated by $n()$. A compound key is still a single key; it can not be disassembled into multiple keys.

There are also absolute keys, that is, local keys prefixed with a '$l::$', where l specifies the reference frame. Absolute keys and local keys comprise value, v, which can be passed as argument in function call, or be stored into or be retrieved from resources.

In addition, a local key may also be prefixed with a variable as in $x.i$, but the variable must be substituted with value before the key can be used in computation.

A cluster is several keys stringed up in a sequence with '::'. By using them one by one, one can recursively enter deeper and deeper into the hierarchy. For example, $m.n::m'$ is a cluster of two keys, where m' is a key to a resource that is local at the scope denoted by key $m.n$. As with keys, cluster can be prefixed with '$l::$' or with a variable.

In Scope language, clusters are mostly used to implement dynamic introspection and invocation return path. For dynamic introspection, only limited form of cluster is needed, r, i.e. clusters composed of at most two keys. Full form of cluster as I and V are used in return path.

2. Expression and Function

The construct for expression is almost the same as in SL language, except that it must be ensured that only keys, not clusters, are used as values in the expression. As variables may appear in an expression as well as names, a function should have all its variables bound.

3. Return, Continuation and Process

Return is a function prefixed with a return path. When an agent returns, the return path is first used for locating the scope where the agent returns, then the function is applied with return value as argument to resume the computation there.

Continuation and Process are the same as in SL language.

4. Scope, Resource and Introspections

In Scope language, resources are modified by attributes; $\uparrow$ means public resource, while $\downarrow$ means private resource. In addition, there is a new kind of resource, i.e. scope resource $n(S(H))$. The H in $S(H)$ means introspections. It is a multiset of introspections. An introspection is of form $l \succ v$, which means that the current scope statically introspects the scope located by v, so that the latter may be used as a reference frame with name l and by agents inside the boundary of the former. Scope $S(H)$ also has an expanded form of $\epsilon R, \epsilon T, \ldots \vdash_l^H P, Q, \ldots$, which is actually an environment with name l and introspections H.

5.4 Transition semantics

In Scope language, computation happens across a hierarchy of scopes. One step of computation may involve multiple entities located far distance apart in the hierarchy. For such a system, convenience is no more possible for us to give the semantics in a reduction system. We should instead choose to present the semantics of Scope language in terms of a labelled transition system.

To make the transition system simple, keys and clusters (i.e. i, v, I, V, k, r) are identified up to the associativity of arbitrary combination of '.' and '::'. Scopes are identified up to the renaming of bound variables as well as to the following structure congruence rules.

Structure Congruence	
$\vdash (r:(k)F)C \equiv \vdash r:(k)(FC)$	(Struct CPS)
$\vdash 0 \equiv \vdash$	(Struct GC)

The transition system of Scope language consists of two parts. One is a collection of on-the-scene transition rules; the other is a collection of propagation rules. In comparison to previous reduction system, the transition system is complicated somewhat. We can note that the number of rules is doubled here.

On-the-scene Transition	
$\epsilon n\langle F\rangle \vdash \xrightarrow{in\overline{\omega}(v,C)} \epsilon n\langle F\rangle \vdash vFC$	(Trans Inv In)
$\epsilon n\langle v\rangle \vdash \xrightarrow{in\overline{\omega}v} \epsilon n\langle v\rangle \vdash$	(Trans Fetch In)
$\epsilon n\langle v\rangle \vdash \xrightarrow{in\overline{\omega}u} \epsilon n\langle u\rangle \vdash$	(Trans Asgn In)
$\vdash V:(v)C \xrightarrow{V_1(v,C)} \vdash$	(Trans Inv Out)
$\vdash V?(x).EC \xrightarrow{V?v} \vdash E[v/x]C$	(Trans Fetch Out)
$\vdash V!(v).EC \xrightarrow{V!v} \vdash EC$	(Trans Asgn Out)
$\vdash \text{fork } E'.EC \xrightarrow{\tau} \vdash E' \varnothing, EC$	(Trans Fork)
$\vdash v((\lambda x)E)C \xrightarrow{\tau} \vdash E[v/x]C$	(Trans Call)
$\vdash v\varnothing \xrightarrow{\tau} \vdash 0$	(Trans End)
$\vdash \xrightarrow{FC@v} \vdash vFC$	(Trans Ret In)
$\vdash vMC \xrightarrow{MC@v} \vdash$ (where $M \neq F$)	(Trans Ret Out)

Propagation

$$\frac{S(H) \xrightarrow{\alpha} S'(H) \quad abs(\alpha) \vee in(\alpha)}{\epsilon n\langle S(H)\rangle \vdash_I \xrightarrow{in(\vdash^{H}\alpha)\vdash_I} \epsilon n\langle S'(H)\rangle \vdash_I} \quad \text{(Trans Prop } \alpha\text{)}$$

$$\frac{S(H) \xrightarrow{\tau} S'(H)}{\epsilon n\langle S(H)\rangle \vdash \xrightarrow{\tau} \epsilon n\langle S'(H)\rangle \vdash} \quad \text{(Trans Prop } \tau\text{)}$$

$$\frac{S(H) \xrightarrow{\alpha|\alpha'} S'(H) \quad val(\widetilde{\alpha}) = val(\alpha')}{S(H) \xrightarrow{\tau} S'(H)} \quad \text{(Trans React)}$$

where $S(H) \xrightarrow{\alpha|\alpha'} S'(H)$ means

$$S(H) \xrightarrow{\alpha} \xrightarrow{\alpha'} S'(H) \text{ or } S(H) \xrightarrow{\alpha'} \xrightarrow{\alpha} S'(H)$$

Let $\omega ::= ? \mid ! \mid :\mid @$, $\pi ::= \omega \mid \overline{\omega}, U ::= I \mid FC \mid I::FC$ and $W ::= V \mid C \mid \epsilon I \ (\epsilon I::F)C$, then we will have $\alpha ::= W\pi W' \mid W\pi \ (W', W'')$ where W is the subject of α and W' and W'' are the objects of α.

- $abs(\alpha) = true$ iff the subject W is prefixed with $I::$
- $in(\alpha) = true$ iff $\overline{\omega}$ is in α
- $val(\alpha)$ removes all ϵ in α
- α change ω in α to $\overline{\omega}$, and $\overline{\omega}$ to ω

The translation function $\epsilon n\langle \vdash^{H}\alpha\rangle \vdash_I$ translates the subject and the objects in α as below.

$$\epsilon n(\vdash^H W)\vdash_I :: = \begin{cases} \epsilon n.I & W = \uparrow I \\ (\epsilon n.I::F)C & W = (\uparrow I::F)C \\ \epsilon n::I & W = \epsilon' I \\ (\epsilon n::I::F)C & W = (\epsilon' I::F)C \\ \epsilon n::U & W = U \\ v::U & W = s::U \wedge s \xrightarrow{} v \in H \\ U & W = s::U \wedge l = s \\ W & W = s::U \wedge (s \xrightarrow{} v \notin H) \wedge l \neq s \\ \varnothing & W = \varnothing \end{cases}$$

5.5 Explanation for the transition system

In a hierarchy of scopes, computation happens with the interleaved occurrences of actions from different agents. Each occurrence of action elicits a transition on every scope that is affected by it (either directly on the scope itself or indirectly on any of its sub-scopes recursively). The directly affected scopes are the scenes of the action. All the scenes of an action comprise its effect range.

As a scope is a reference frame, a transition on the scope is actually an observation in the reference frame of the action causing the transition. Within different reference frames, we will have different observations of the action.

A transition in Scope language is of the form:

$$S(H) \xrightarrow{\eta} S'(H)$$

Intuitively, it means that after the occurrence of an action that is observed as event η within the reference frame of $S(H)$, scope $S(H)$ evolves to $S'(H)$. The event η has the following forms:

• The closed event τ. An event τ is an observation of an action whose effect range is contained by the boundary of the current reference frame.

• The open event α. An event α is an observation of an action whose effect range is beyond the boundary of the current reference frame. α is an expression of the form:

$$\alpha :: = W\pi W' \mid W\pi(W', W'')$$

Within a α expression, if π is ω, the α is an in event; otherwise, it is an out event.

If the transition is on the scenes of the action, it will be a on-the-scene transition. On-the-scene transitions are transitions where an action originates or destines. All other transitions are derived from them by propagation rules, which are on scopes indirectly affected by the action.

In Scope language, there are eleven on-the scene transition rules. Trans Call and Trans End are observation of actions internal to an agent. Trans Fork is observation of an action local to the scope of the agent. The rest four pairs of transitions are observation of action with distributed effect range.

Trans Ret In is the observation of the arrival of an agent at the destination scope, while Trans Ret Out is the observation of the leave of an agent at the departure scope. They form a pair of observations on the "return" action of an agent.

In the same way, Trans Inv In/Out, Trans Asgn In/Out, and Trans Fetch In/Out are pairs of observations for invocation, assignment and fetch instruction respectively, which are interactions between an agent and a resource. Among them, the transitions of in event are observations within the reference frame of the resource, while the transitions of out event are observations within the reference frame of the agent.

For internal and local actions, we could only observe closed events. For distributed actions we may as well observe open events. Open events are propagated up the hierarchy by rule Trans Prop α. Closed events are propagated up the hierarchy by rule Trans Prop τ. Closed event and in event can be propagated arbitrarily high in

the hierarchy, whereas out event has a ceiling on its propagation: it can not propagate out of the innest boundary containing the action's effect range. This ceiling is enforced by the condition, $abs(\alpha) \vee in(\alpha)$, in the Trans Prop a rule.

When two complementary open events meet in a common reference frame, they will react according to Trans React and resolve into a closed event. Two open events are complementary if and only if one is an in event; the other is an out event, and their subject and objects match with each other. That is, $val(\overline{\alpha}) = val(\alpha')$.

When an event is propagated up the hierarchy into a new reference frame, it is observed as a different event, i.e. the subject and objects of the event need to be translated. So we have defined a function, $\langle n \vdash^H \alpha \rangle \vdash_t$, which translates an event α observed within the current scope $\vdash^H$ to the corresponding event observed at the parent scope $\vdash_t$.

Finally, one thing to note is that in a distributed action only value and continuation are passed between scenes. They are carried in the expression of open events, and are usually the objects of the event expression (in the case of Trans Ret In/Out they may be the subject, too). Therefore, it must be ensured that the value passed out of one scene is translatable to a value within another scene.

5.6 Examples

Programming in Scope language is direct and interesting. In this section, we give two program fragments to illustrate the style of programming in Scope language.

A Cell In Ref. [3], Hennessy uses a cell example to illustrate programming in $D\pi$. A cell is a place for storing values; channel p is used to put value into the cell, while g is used to get value out of the cell.

$System \Leftarrow l[Cell(v)] \mid h[User]$

$Cell(n) \Leftarrow (\nu s)s!\langle n\rangle$

$\quad \mid *g?(y)s?(x)(s!\langle x\rangle \mid y.ret!\langle x\rangle)$

$\quad \mid *p?(y,v)s?(x)(s!\langle v\rangle \mid y.ack:\langle\rangle)$

$User \Leftarrow l.p!\langle h,0\rangle \mid ack?()(l.g!\langle h\rangle ret?(x)print!\langle x\rangle)$

The cell has an internal channel s in which the value is stored. When getting a value, the user needs to furnish a return address, which is used by the cell to send back the value after the retrieval from s. When putting the value, the user furnishes a new value in addition to a return address, but the address is only for making an acknowledge to the user (after the value is updated).

In comparision with it, we give our implementation of the cell example as follows.

$System \Leftarrow m\langle Cell(v)\rangle, n\langle User\rangle \vdash,$

$Cell(X) \Leftarrow \downarrow c\langle X\rangle, \uparrow g\langle(\lambda)c?(x).x\rangle, \uparrow p\langle(\lambda x)c!(x)\rangle \vdash$

$User \Leftarrow \uparrow a\langle\rangle \vdash s::m.p:(a)(\lambda)(s::m.g:()(\lambda x)x) \varnothing_{print}$

It is obvious that our program is simpler than the $D\pi$ program. The programming style is also much more straightforward, for the user needs not to explicitly pass any return address in interaction with the cell.

A Cell Market Using $D\pi$, Hennessy also implements a cell server. A cell server is a place where users could obtain fresh cells. Below, we have adapted the example and implemented a cell market where a number of cells are advertised on BBS for sale, and a number of users bid to buy them.

$\text{Market} \Leftarrow \ldots, m_i\langle Cell_i(v_i)\rangle, \ldots$

$\quad \ldots, n_j\langle User_j\rangle \ldots$

$\quad BBS\langle NA\rangle, NA\langle bid\langle(\lambda x)x.find:()\rangle \vdash\rangle \vdash,$

$Cell_i(X) \Leftarrow \downarrow c\langle X\rangle, \uparrow g\langle(\lambda)c?(x).x\rangle, \uparrow p\langle(\lambda x)c!(x)\rangle,$

$\quad \downarrow job\langle advertise\rangle, \downarrow bidder(), \uparrow status\langle vacant\rangle,$

$\quad \downarrow advertise\langle(\lambda)(s::BBS)!(m_i).job?(x).x:()\rangle,$

$\downarrow selection\langle(\lambda)(s::BBS)!\ (NA).bidder?\ (x).(x.notification)!\ (x.win)\rangle$

$\uparrow bid\langle(\lambda x)bidder!\ (x).job!\ (selection).(x.wait).(m_i)\rangle,$

$\uparrow vacant\langle(\lambda x)(x.notification)?\ (y).y;(m_i)\rangle,$

$\uparrow occupied\langle(\lambda x)x.find()\rangle$

$\vdash job?\ (x).x;()\not\subset$

$User_i \quad \Leftarrow \uparrow a(), \uparrow notification\langle wait\rangle,$

$\downarrow find\langle(\lambda)(s::BBS)?\ (x).(x.bid);(n_j)\rangle,$

$\uparrow wait\langle(\lambda x)(x.status)?\ (y).y;()\rangle,$

$\uparrow win\langle(\lambda x)(x.status)!\ (x.occupied).x\rangle$

$\vdash find;()(\lambda x)(x.p);(a)(\lambda)((x.g);()(\lambda y)y)\not\subset_{print},$

In a bidding, the cell is responsible to select a winning bidder, and to notify him about the win. The winner will in turn occupy the cell, and make other bidders give up. The giver-ups will then re-start to find new cells for bidding.

6 Summary and Future Work

Our language is quite unique in its formal treatment of the notions of resource, agent and scope. In other formalisms, resources do not appear as independent syntactical entities. For example in Ambient, ambient is process and process is ambient; but the notion of resource does not exist. In Seal, channels exist only semantically but have no syntactical embodiment. In our formalism, we use separate syntactic entities to represent agent, resource and scope. It brings a proliferation of constructs in our formalism and poses some challenges for the formal system. However, as we have already said, we are not looking for semantical or syntactical minimality. Our language is intended as a basis for real programming language. It should be able to directly represent most constructs used in programming, and not too far away from an implementable model of programming language.

Besides the differences in goals, the basic model used in our language is distinct from other work as well. The distinction is not so much in the uniqueness of individual features than in the novelty of combination of features.

· In our model, we have adopted the hierarchical organization of scopes as in Seal and Ambient instead of the flat set organization as in $D\pi$. But scopes in our language are not directly composable. They should first form resources, and then be used in composing other scopes.

· The keys in our language are statically bound to resources, instead of dynamical bound as in most other calculi. It necessitates a translation function in our language to translate keys when passed between different scopes, so that they keep referencing the same resource.

· Mobility in our model is only supported in the form of agent mobility as in $D\pi$, instead of location movement as in Seal and Ambient. It is a restricted form of mobility, but on the other hand, the absence of scope mobility makes implementation easier, as it will not need to implement code mobility. So in this respect, our model is more conventional than other work.

The formal system presented in this paper provides just a starting point for our future research, where there are many interesting issues untouched yet. However, to be short, here we just mention one of the most important thing missing from the current language: the type system. A type system shall be important for Scope language. For example, our syntax currently allows to write programs that make invocations to data resources, which in fact have no defined meaning in our language. If there is a type system and type is assigned to resources and keys, syntax checking would be able to detect the illegal usage.

A type system also helps in some other aspects significantly, such as typing a name into a read or a write capability, or typing a reference frame to restrict the resources accessable inside it, etc. Type system and some other

issues around Scope language are the subjects we are actively working on.

References:

[1] Cardelli, L, Gordon, A.D. Mobile ambients. In Nivat, M., ed. Proceedings of the FoSSaCS'98. Volume 1378 of LNCS, Dpringer, 1998. 140~155.

[2] Fournet, C., Gonthier, G., Lévy, J. J., *et al.* A calculus of mobile agents. In: Ugo, M., Vladimiro, S., eds. Proceedings of the CONCUR'96. Volume 1119 of LNCS, Springer, 1996. 406~421.

[3] Hennessy, M., Riely, J. Resource access control in systems of mobile agents. In: Uwe, N., Pierce, B. C., eds. Proceedings of the HLCL'98. Volume 16.3 of ENTCS, Elsevier Science Publishers, 1998. 3~17.

[4] Vitek, J., Castagna, G. Seal: a framework for secure mobile computations. In: Bal, H.E., Boumediene, B., Cardelli, L., eds. Internet Programming Language, Number 1686 in Lecture Notes in Computer Science Springer-Verlage, 1999. 44~77.

[5] Amadio, R.M. On modelling mobility. Theoretical Computer Science, 2000,240(1):146~176.

[6] Cardelli, L. A language with distributed scope. Computing Systems, 1995,8(1):27~59.

[7] White, J.E. Mobile agent. In: Bradshaw, J. ed. Software Agent. AAAI Press/The MIT Press, 1997. 437~472.

[8] Boudol, G. The π-calculus in direct style. Higher-Order and Symbolic Computation, 1998,11:177~208.

[9] Agha, G.A., Mason, I.A., Smith, S.F., *et al.* A foundation for actor computation. Journal of Functional Programming, 1997,7(1):1~72.

[10] Berry, G., Boudol, G. The chemical abstract machine. Theoretical Computer Science, 1992,96:217~248.

一种基于分布式资源域理论的形式化语言

王 翔, 黄 涛, 冯玉琳

(中国科学院 软件研究所 计算机科学开放研究实验室,北京 100080)

摘要: 介绍了一种带有Agent移动的分布计算形式化语言——资源域语言.与分布移动计算中的同类工作相比较,它没有采用基于π-演算进行扩充的方法,而是以一种带资源的类λ-演算为核心发展而来.这使得它可以直接支持内存单元式的资源,而不是间接地通过进程/通道来模拟.进一步地,资源域语言还对"位置"的概念进行了特殊的处理,产生了"域"的概念.域和内存资源结合在一起使得资源域语言成为对现有同类工作的一种有益补充,提供了描述分布移动系统的一种新途径.其主要特点包括:易于实现和更接近实际语言(如Obliq,Telescript)的程序设计模型.

关键词: 移动Agent;资源;位置;资源域;形式化语言

中图法分类号: TP311 **文献标识码:** A