Chapitre 5

PROTOTYPES ET ACTEURS

Ce chapitre présente deux variations importantes des idées de base des langages à objets. Les *langages de prototypes* font disparaître la différence entre classes et instances en introduisant la notion unique d'objet prototype. Ils remplacent la notion d'héritage par celle de délégation. Les *langages d'acteurs* généralisent la notion d'envoi de message pour l'adapter à la programmation parallèle : l'envoi de message n'est plus une invocation de méthode, mais une requête envoyée à un objet.

5.1 LANGAGES DE PROTOTYPES

Les langages à objets que nous avons présentés jusqu'à présent étaient tous fondés sur les notions de classe, d'instance et d'héritage. Ces trois notions induisent deux relations entre les entités du langage : la relation d'instanciation entre un objet et sa classe, et la relation d'héritage entre une classe et sa classe de base. Pour distinguer ces langages à objets classiques des

langages de prototypes, nous appellerons les premiers *langages de classes*.

Les premiers travaux sur les langages de prototypes datent de 1986 ; ils sont dus à Henry Lieberman du MIT, qui à la même époque a aussi travaillé sur les langages d'acteurs. Le langage qui a inspiré cette présentation s'appelle Self, créé et développé depuis 1987 par David Ungar et Randall Smith à l'université de Stanford. Il représente l'étape la plus avancée dans le domaine des langages de prototypes.

Prototypes et clonage

Dans un langage de prototypes, il n'y a pas de différence entre classes et instances : tout objet est un *prototype* qui peut servir de modèle pour créer d'autres objets. L'opération qui permet de créer un nouvel objet à partir d'un prototype s'appelle le *clonage*, et consiste à recopier l'objet cloné.

Dans un prototype, l'état et le comportement sont confondus : il n'y a pas de différence entre champs et méthodes, appelés indistinctement *cases* (« slots » en anglais). Pour accéder au champ *x*, un prototype s'envoie le message *x*. Pour modifier le champ *x*, il s'envoie le message *x:* avec la nouvelle valeur en argument.

Nous déclarons un prototype par une liste de couples nom de case / valeur, entourée d'accolades. Les champs ont pour valeur une expression tandis que les méthodes ont pour valeur un bloc (noté entre crochets, comme en Smalltalk). Un prototype de pile aura ainsi l'aspect suivant :

```
Pile ← {
    pile                Tableau cloner.
    sommet              0.
    Empiler: unObjet    [ sommet: (sommet + 1).
                          pile en: sommet mettre: unObjet ].
    Dépiler             [ sommet: (sommet - 1) ].
    Sommet              [ ↑ pile en: sommet  ].
}
```

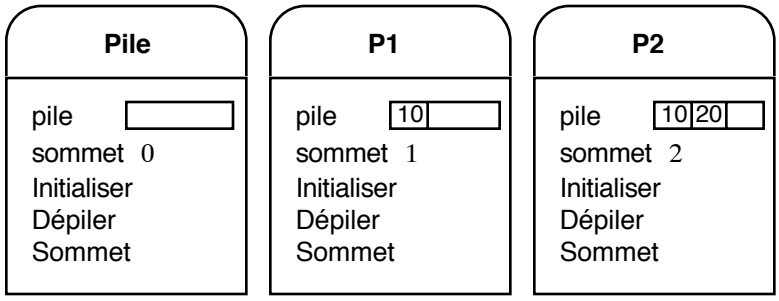

Figure 21 - Le prototype *Pile* et deux clones

Les cases *pile* et *sommet* sont des champs. Leurs valeurs correspondent aux valeurs initiales à la création du prototype. Les messages d'accès *pile* et *sommet*, et les messages d'affectation *pile:* et *sommet:* sont implicitement créés. Les autres cases sont des méthodes. Comme tous les accès aux champs se font par messages, la pseudo-variable *self* est le receveur implicite des messages. La méthode *Dépiler* pourrait s'écrire sous la forme :

```
Dépiler          [ self sommet: (self sommet - 1) ].
```

Dans cet exemple, *Pile* est un prototype, donc un objet directement utilisable. Nous allons utiliser *Pile* comme un modèle pour créer et manipuler deux piles (figure 21) :

```
p1 ← Pile cloner.
p1 Empiler: 10.
x ← p1 Sommet.              -- x vaut 10
p2 ← p1 cloner.             -- p2 contient déjà 10
p2 Empiler: 20.
```

Dans un langage de classes, une classe contient une *description* de ses instances. Dans un langage de prototypes, tout objet est un *exemplaire* qui peut être reproduit par clonage. Comme on le voit dans l'exemple ci-dessus, ce mécanisme simplifie l'initialisation des objets : il suffit d'initialiser correctement le prototype qui sert de modèle. À titre de comparaison, Smalltalk exige de redéfinir la méthode

d'instanciation définie dans la métaclasse ; avec les langages à objets typés, il faut introduire des mécanismes spécifiques tels que les constructeurs de C++.

La délégation

Le clonage consiste en la duplication exacte d'un prototype dans son état courant. Cela signifie que l'état et le comportement sont copiés et qu'il n'y a pas de lien entre l'objet qui sert de modèle et l'objet résultant du clonage, comme illustré dans la figure 21. Il n'y a donc pas de possibilité de partage entre les objets. Dans les langages de classes, le partage existe à deux niveaux :

- par le lien d'instanciation entre un objet et sa classe, toutes les instances d'une classe partagent le même comportement, décrit dans la classe.
- par le lien d'héritage entre une classe et sa superclasse, les classes dérivées partagent les descriptions contenues dans leurs superclasses.

Dans les langages de prototypes, un mécanisme unique, la *délégation*, permet à des objets de partager des informations. Un objet peut déléguer à un autre objet, appelé *parent*, les messages qu'il ne comprend pas. Dans l'exemple de la pile, nous allons partager les méthodes *Empiler*, *Dépiler* et *Sommet* en les mettant dans un prototype à part, qui deviendra le parent de tous les clones de la pile. Lorsque l'un de ces messages sera envoyé à un clone de la pile, il sera délégué à son prototype, dans ce cas *protoPile*.

```
protoPile ← {
    Empiler: unObjet      [ sommet: (sommet + 1).
                            pile en: sommet mettre: unObjet ].
    Dépiler               [ sommet: (sommet - 1) ].
    Sommet                [ ↑ pile en: sommet  ].
}
```

Le prototype de la pile s'écrit maintenant comme suit :

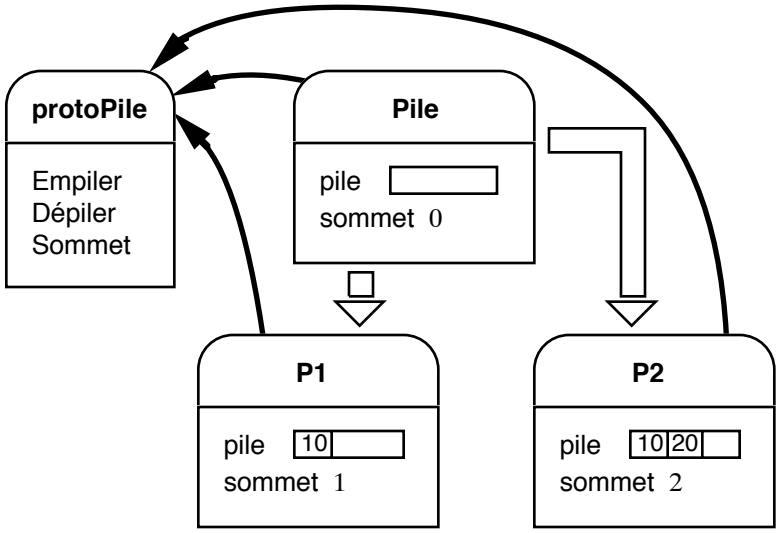

Figure 22 - Partage de méthodes avec les prototypes.
Les flèches noires indiquent le parent (qui est un champ).
Les flèches blanches indiquent les opérations de clonage

```
Pile ←{
    parent      protoPile.
    pile        Tableau cloner.
    sommet      0.
}
```

Lorsque l'on clone un prototype, les objets créés ont le même parent que leur prototype. Dans notre exemple, les clones de *Pile* ont pour parent *protoPile*. L'exemple d'utilisation vu plus haut est toujours valable, mais le schéma des objets à l'exécution change, comme le montre la figure 22. Lorsque l'on envoie le message *Empiler* à *p1*, il est délégué à son parent *protoPile*.

Dans cet exemple, le couple *(Pile, protoPile)* joue le rôle d'une classe dans un langage de classes. Le lien d'instanciation entre un objet et sa classe est réalisé par la délégation. L'instanciation se fait par clonage, mais on pourrait aisément

simuler le mécanisme des langages de classe en définissant dans *protoPile* la méthode *Nouveau*, qui aurait pour valeur

```
[ ↑ Pile cloner].
```

Cet exemple montre également pourquoi l'accès aux champs se fait par message : dans les corps des méthodes de *protoPile*, on fait référence à des champs (*sommet*, *pile*) qui sont déclarés dans *Pile*, donc inconnus de *protoPile*.

Dans la suite, nous utiliserons le terme « classe » pour parler d'un prototype qui contient exclusivement des méthodes. Il faut néanmoins garder à l'esprit qu'une classe n'est pas une notion distincte dans les langages de protoypes.

Simuler l'héritage avec la délégation

Nous allons maintenant illustrer l'utilisation de la délégation pour simuler l'héritage. Reprenons pour cela l'exemple des Tours de Hanoi. Une tour est une pile qui doit contrôler la taille des objets empilés. On crée donc un prototype *Tour* qui a pour parent le prototype *protoTour*, qui a lui-même pour parent *protoPile*.

```
protoTour ← {
    parent          protoPile.
    Empiler: unObjet      [
        unObjet < Sommet
            siVrai: [ parent Empiler: unObjet ]
            siFaux: [ "Empiler: objet trop grand" Écrire ]
    ].
}
Tour ← (Pile cloner) parent: protoTour.
```

Nous avons défini dans ce nouveau prototype une méthode *Empiler:* qui, selon la taille de l'objet, demande à son parent de réaliser l'empilement ou affiche un message d'erreur. Le prototype *Tour* est créé par clonage du prototype *Pile*, en changeant son parent. L'utilisation de *Tour* est illustrée par l'exemple suivant et la figure 23.

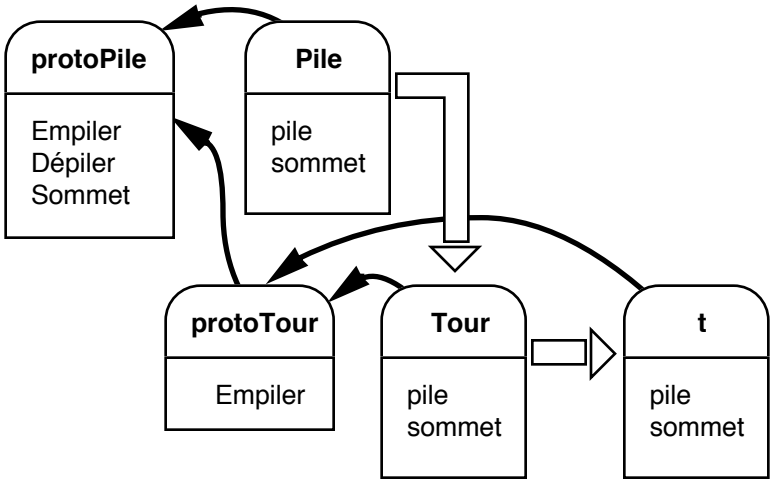

Figure 23 - Simulation de l'héritage par la délégation.
Les flèches ont la même signification que pour la figure 22

```
t ← Tour cloner.
t Empiler: 10.
t Empiler: 20.                -- Empiler: objet trop grand
```

L'exemple de la pile que nous venons de présenter permet de faire le parallèle entre les langages de prototypes et les langages de classes. Mais l'intérêt des prototypes est d'étendre les possibilités des langages de classes. Les prototypes permettent ainsi, entre autres, de créer des objets dotés de comportements exceptionnels, d'avoir des champs calculés, et de faire de l'héritage dynamique. Nous allons maintenant illustrer ces différentes possibilités.

Comportements exceptionnels

Dans l'exemple de la tour, si notre application utilise une seule tour, on peut créer un objet qui a le comportement d'une tour sans pour autant créer la classe correspondante. Il suffit de redéfinir la case *Empiler:* dans l'objet lui-même :

```
t ←{
    parent          protoPile.
    pile            Tableau cloner.
    sommet          0.
    Empiler: unObjet        [
        unObjet < Sommet
            siVrai: [ parent Empiler: unObjet ]
            siFaux: [ "Empiler: objet trop grand" Écrire ]
    ].
}

t Empiler: 10.
t Empiler: 20.                  -- Empiler: objet trop grand
```

Cet exemple montre comment créer des objets avec des comportements exceptionnels. Les objets *vrai* et *faux* de Smalltalk sont un autre exemple d'application : là où nous avions créé deux classes *Vrai* et *Faux*, avec chacune une instance unique, il nous suffit de créer deux prototypes *vrai* et *faux*, contenant chacun une version de la méthode *siVrai:siFaux:*. On peut noter que de tels objets peuvent être clonés, les clones disposant également du comportement exceptionnel.

Une autre application des comportements exceptionnels est la mise au point de programme. Si l'on veut suivre le comportement d'un objet précis, on peut définir une méthode dans l'objet de la façon suivante (nous utilisons la méthode *ajoute:* qui permet d'ajouter des cases dans un objet) :

```
p ← Pile cloner ajoute: {
    Empiler: unObjet        [
        "p Empiler" Écrire.
        parent Empiler: unObjet
    ]
}
```

Tout empilement sur *p* provoquera un message. Dans un langage de classes, l'ajout d'une trace dans la méthode *Empiler* de la classe *Pile* provoquerait l'écriture du message pour tout objet de la classe *Pile*.

Champs calculés

L'accès aux champs d'un prototype par envoi de message permet de définir des champs dont la valeur est calculée et non pas stockée dans l'objet. Pour illustrer cela, définissons un prototype *protoPoint* qui contient des méthodes de manipulation de points, et deux prototypes de points : l'un dont la position est stockée en coordonnées cartésiennes (*Cartésien*), l'autre en coordonnées polaires (*Polaire*) :

```
protoPoint ← {
    Écrire    [ x Écrire. ", " Écrire. y Écrire ].
    +: p      [ ↑ cloner x: (x + p x) y: (y + p y) ].
}
```

La méthode +: crée un nouveau point dont les coordonnées sont la somme des coordonnées du receveur et de l'argument.

```
Cartésien ← {
    parent    protoPoint.
    x         0.
    y         0.
}
```

```
Polaire ← {
    parent    protoPoint.
    rho       0.
    thêta     0.
}
```

Malheureusement, le prototype *Polaire* est inutilisable car les méthodes de *protoPoint* utilisent les champs *x* et *y*. Pour remédier à cette situation, il suffit d'ajouter les méthodes *x*, *y*, *x:* et *y:* à Polaire pour simuler les champs absents, grâce aux équations suivantes :

$$x = \rho \cos \theta \qquad \rho = \sqrt{(x^2 + y^2)}$$
$$y = \rho \sin \theta \qquad \theta = \text{atan}\ (y / x)$$

De l'extérieur, tout ce passe comme si *Polaire* avait les champs *x* et *y*, qui sont en réalité calculés à partir des coordonnées polaires stockées dans l'objet.

```
Polaire ← {
    parent    protoPoint.
    rho       0.
    thêta     0.
```

```
    x          [ ↑ rho * thêta cos ].
    y          [ ↑ rho * thêta sin ].
    x: val     [ rho: (val * val + y * y) sqrt. thêta: (y / val) atan ].
    y: val     [ rho: (x * x + val * val) sqrt. thêta: (val / x) atan ].
}
```

On pourrait de façon similaire ajouter les champs calculés *rho* et *thêta* au prototype *Cartésien*. Cela permettrait d'utiliser les coordonnées les plus adéquates dans les méthodes de *protoPoint*.

Héritage dynamique

Le parent d'un prototype est une case similaire aux autres, à l'exception de son rôle particulier pour la délégation lors de l'envoi de messages. Rien n'interdit donc de modifier la valeur de la case qui contient le parent d'un objet après la création de celui-ci : c'est l'*héritage dynamique*. Ainsi, en utilisant les définitions de *Pile* et de *Tour* vues plus haut, on peut transformer une tour en pile en affectant sa case *parent* :

```
t ← Tour cloner.
t Empiler: 10.
t Empiler: 20.          -- Empiler: objet trop grand
t parent: Pile.         -- la tour devient une pile
t Empiler: 20.          -- OK
```

Les applications de cette technique sont multiples. Par exemple, il arrive que la classe d'un objet ne puisse être déterminée complètement à sa création, ou qu'elle soit amenée à changer lors de la vie de l'objet. L'héritage dynamique permet de préciser la classe au fur et à mesure des connaissances acquises. Par exemple, dans un système graphique, des opérations entre objets graphiques permettent de créer de nouveaux objets. Si un objet ainsi créé se trouve être un objet régulier comme un rectangle, il peut changer de parent pour utiliser des méthodes plus efficaces que celles définies sur un objet quelconque. Inversement, si un rectangle est transformé par une rotation, il devient un polygone, et doit changer de parent en conséquence.

L'héritage dynamique est également utile lors de la mise au point d'un programme : pour observer un objet, on lui affecte comme parent un prototype qui trace les opérations effectuées. On peut également tester une nouvelle implémentation d'une classe en affectant, en cours d'exécution, la nouvelle classe au parent d'un objet.

Conclusion

En abolissant les différences entre classe et instance, et en unifiant les champs et les méthodes, les langages de prototypes ouvrent de nouvelles portes aux langages à objets à liaison dynamique.

De façon assez surprenante, l'implémentation d'un langage de prototypes peut être plus efficace que celle d'un langage tel que Smalltalk. Ceci nécessite néanmoins la mise en œuvre de techniques assez complexes, qui consistent pour l'essentiel à garder dans des caches les résultats des recherches de méthodes pour optimiser les envois de messages, et à compiler différentes versions d'une méthode pour des contextes d'appels différents.

Il n'en reste pas moins que les langages de prototypes sont des langages non typés, donc sans aucun contrôle de la validité d'un programme avant son exécution. Autant il est envisageable d'ajouter des déclarations de type dans un langage comme Smalltalk, autant cela est illusoire dans un langage de prototypes. En effet, la notion de type, qui correspond à celle de classe dans un langage de classes, n'a pas vraiment d'équivalent dans un langage de prototypes. Si l'on considère que le type d'un objet est son parent, tout typage statique est impossible car le type de l'objet peut changer durant sa vie. Par ailleurs, la structure d'un objet peut aussi changer par ajout de cases, de telle sorte qu'utiliser la structure d'un objet comme type est également impossible.

En l'absence de moyens de vérification statique des programmes, les langages de prototypes restent donc réservés essentiellement au prototypage…

5.2 LANGAGES D'ACTEURS

Les langages d'acteurs sont nés au MIT des travaux de Carl Hewitt dans les années 70 avec le langage Plasma. Au début des années 80 Henry Liebermann, du MIT, a développé ACT1, puis Akinori Yonezawa de l'Institut de Technologie de Tokyo a introduit ABCL/1.

L'objet des langages d'acteurs est de fournir un modèle de calcul parallèle fondé sur des entités indépendantes et autonomes communiquant par messages. Ces entités, appelées *acteurs*, sont composées d'un état et d'un filtre. L'état est constitué de variables locales et de références à d'autres acteurs, tandis que le filtre est une suite de modèles de messages auxquels l'acteur peut répondre. Chaque acteur est autonome : lorsqu'un message arrive, il vérifie s'il correspond à un modèle de son filtre. Si c'est le cas, l'acteur receveur exécute le bloc d'instructions correspondant. Sinon, l'acteur délègue le message à un autre acteur, appelé son *mandataire* (« proxy » en anglais). Ce mécanisme de délégation est similaire à celui des langages de prototypes, et nous ne reviendrons pas dessus.

Les seules actions que peut effectuer un acteur sont l'*envoi de message*, la *création* de nouveaux acteurs, et sa *transformation* en un autre acteur. L'envoi de message est asynchrone : l'acteur ne se soucie pas de ce qu'il advient des messages qu'il envoie. Cet envoi asynchrone introduit le parallélisme de manière naturelle : l'acteur continue son activité pendant que le message envoyé est traité par son destinataire. Comme chaque acteur est séquentiel, il dispose d'une boîte aux lettres dans laquelle sont stockés les messages qui arrivent pendant qu'il traite un message.

La création d'acteur est simple : un acteur peut créer un autre acteur, dont il spécifie le mandataire, l'état, et le filtre. La transformation d'un acteur est similaire à sa création, à la différence que le nouvel acteur remplace le précédent, c'est-à-dire qu'il récupère sa boîte aux lettres. Dans le modèle introduit par Gul Agha, un acteur peut se transformer avant d'avoir

terminé le traitement du message en cours. Dans ce cas, un acteur peut traiter plusieurs messages en parallèle : il lui suffit, lorsqu'il reçoit un message, de commencer par spécifier son remplaçant. Celui-ci pourra immédiatement traiter le prochain message en attente. La possibilité de traiter des messages en parallèle interdit de modifier l'état de l'acteur. Ceci justifie la nécessité de fournir explicitement un remplaçant, qui est généralement une copie modifiée de l'acteur initial.

L'envoi asynchrone de messages, s'il introduit naturellement le parallélisme, pose un problème : comment envoyer un message et obtenir un résultat en retour ? La réponse consiste à transmettre une continuation avec le message. Une *continuation* désigne l'acteur auquel le receveur d'un message devra transmettre sa réponse. Un acteur peut recevoir la réponse d'un message en se mentionnant comme continuation du message, mais il peut aussi mentionner un autre acteur qui saura traiter le résultat mieux que lui.

Considérons par exemple les trois acteurs suivants : le *lecteur* lit des expressions au clavier, l'*évaluateur* évalue des expressions, et l'*imprimeur* affiche des valeurs. Le lecteur, lorsqu'il a lu une expression, envoie un message à l'évaluateur avec comme contenu l'expression à évaluer et comme continuation l'imprimeur. Ainsi, l'évaluateur transmettra directement le résultat à afficher à l'imprimeur. Dans un modèle classique, le lecteur demanderait l'évaluation à l'évaluateur, recevrait la réponse, et la transmettrait à l'imprimeur.

Avec cet exemple, on pourrait penser que la continuation est inutile, puisque l'évaluateur envoie toujours sa réponse à l'imprimeur. Ce n'est pas le cas : l'évaluateur peut, si l'expression est complexe, s'envoyer des messages avec des sous-expressions à évaluer, en se spécifiant comme sa propre continuation. Un autre acteur, qui lit par exemple dans un fichier, peut envoyer des messages à l'évaluateur avec comme continuation un imprimeur qui écrit dans un fichier de sortie (figure 24).

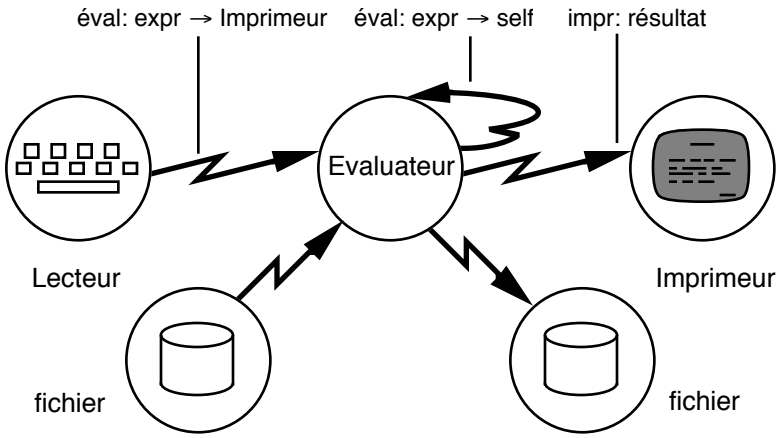

Figure 24 - Envois de messages avec continuations

Programmer avec des acteurs

Pour les exemples de cette section, nous avons adapté la syntaxe utilisée précédemment pour les prototypes. Un acteur est décrit par son nom, éventuellement suivi de son état et de son filtre. Le filtre est un ensemble de couples <modèle de message / action>. Un modèle est un nom de message, avec des arguments et une continuation éventuelle, indiquée par une flèche « → ». La création d'acteurs et l'envoi de message ont la forme suivante :

```
créer acteur (état1, état2, … étatn)
acteur msg1: arg1 msg2: arg2 … msgn: argn → continuation
```

La figure 25 montre la représentation d'un acteur : la flèche horizontale représente la vie de l'acteur, les lignes brisées représentent les envois de messages et les lignes pointillées représentent les créations d'acteurs.

Lorsqu'un acteur reçoit un message, les modèles de son filtre sont comparés au message reçu. Le modèle qui ressemble le plus au message reçu est choisi, et l'action correspondante est

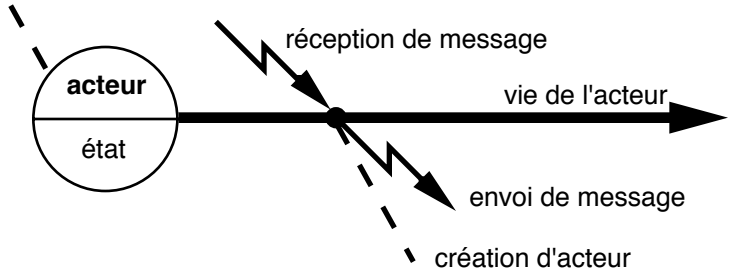

Figure 25 - Représentation d'un acteur

activée. Si aucun modèle ne convient, le message est transmis au mandataire, s'il y en a un ; sinon il y a erreur.

Programmer avec des acteurs exige d'oublier tout ce que l'on sait de la programmation pour apprendre de nouvelles techniques spécifiques du parallélisme. Prenons l'exemple simple de la factorielle, que chacun sait écrire sous la forme d'une fonction récursive. Voici comment on réalise le calcul d'une factorielle avec des acteurs :

```
acteur    factorielle
filtre
    fact: 0 → r    [ r envoie: 1].
    fact: i → r    [ cont ← créer  mult (i, r).
                     self fact: (i - 1) → cont].

acteur    mult
état      val rec
filtre
    envoie: v      [ rec envoie: (v * val)].
```

L'acteur *factorielle* a deux modèles de messages, qui concernent tous les deux le message *fact:* avec une continuation. Le premier modèle ne reconnaît le message *fact:* que lorsque son argument est nul ; le second reconnaît les autres messages *fact:*. L'acteur *factorielle* utilise un autre acteur, *mult*, pour l'aider à faire son calcul. L'état de *mult* est constitué par un entier *val* et un acteur *rec*. Lorsqu'il reçoit le message *envoie:*, il

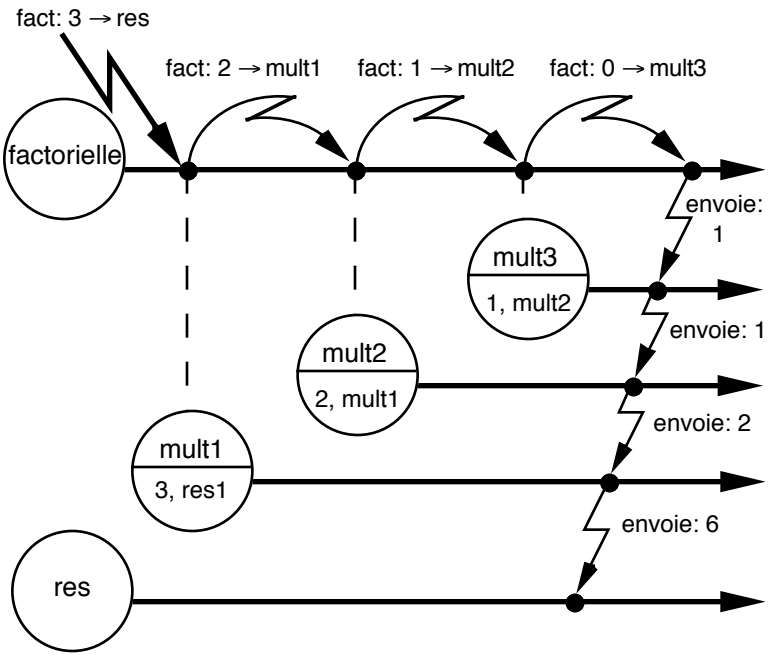

Figure 26 - Calcul de la factorielle

renvoie le message *envoie:* à l'acteur *rec*, avec comme argument le produit de *val* et de la valeur reçue. En d'autres termes, *mult* réalise une multiplication et transmet le résultat à un acteur qui a été spécifié à sa création.

Lorsque l'acteur *factorielle* reçoit un message lui demandant de calculer la factorielle de *i* et de transmettre le résultat à la continuation *r*, il commence par créer un acteur *mult*. Cet acteur multipliera par *i* la valeur qui lui sera transmise, et enverra le résultat à *r*. Ensuite, l'acteur *factorielle* s'envoie un message lui demandant de calculer la factorielle de *i-1* et de transmettre le résultat à l'acteur *mult* qu'il vient de créer. Cet acteur multipliera donc la factorielle de *i-1* par *i*, produisant le résultat escompté. Lorsque *factorielle* doit calculer la factorielle de 0, il envoie directement 1 à la continuation du message, ce qui

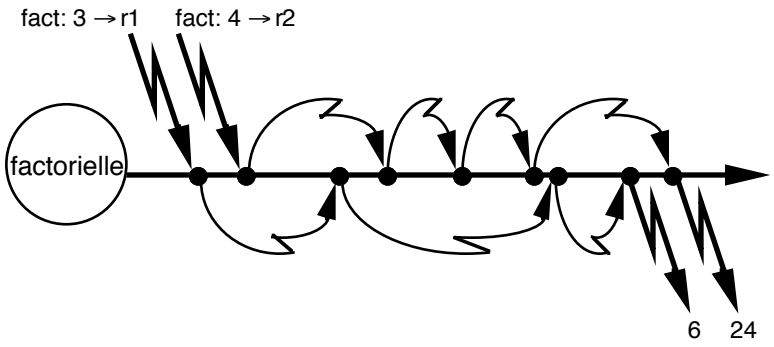

Figure 27 - Calcul simultané de plusieurs factorielles

termine le calcul en évitant l'envoi infini de messages de *factorielle* à lui-même.

La figure 26 visualise le calcul d'une factorielle. Il y a création d'un ensemble d'acteurs $mult_i$, un pour chaque étape du calcul. Ces acteurs sont inactifs jusqu'à réception du message *envoie:*. Le calcul se déclenche en chaîne lorsque *factorielle* envoie le message *envoie:* à $mult_3$.

Le modèle des acteurs nous a obligé à transformer la récursion en un ensemble d'acteurs qui représente le déroulement du calcul. Dans cet exemple, le calcul est strictement séquentiel. Néanmoins, l'acteur *factorielle* est capable de calculer plusieurs factorielles simultanément. En effet, observons ce qui se passe lorsqu'il reçoit deux messages *fact:* (figure 27) : les deux calculs s'enchevêtrent sans se mélanger, car les messages transportent l'information suffisante pour le calcul qu'ils sont en train d'effectuer, sous la forme de continuations.

Il existe d'autres techniques de programmation avec des acteurs, que nous ne détaillerons pas ici, à part la jointure de continuations dont nous donnerons un exemple plus loin.

Envoi de messages

Les langages d'acteurs apportent aux langages à objets une nouvelle vision de l'envoi de message. En fait, il n'y a que dans les langages d'acteurs que le terme d'*envoi de message* est correct : dans les autres langages, il s'agit d'invocation de procédures ou fonctions. La communication asynchrone, que nous avons utilisée jusqu'à présent, n'est pas la seule disponible dans les langages d'acteurs.

ABCL/1, par exemple, dispose de deux autres types de communication. Le premier, la communication synchrone, correspond à l'envoi de message avec attente de réponse. Dans ce cas la continuation est obligatoirement l'émetteur du message. La communication synchrone bloque l'émetteur jusqu'à réception du message. L'autre type de communication d'ABCL/1, la communication anticipée, permet de lever cette contrainte. Au lieu d'être bloqué dans l'attente de la réponse, l'acteur continue à fonctionner. Pour savoir si la réponse est disponible, il interroge le receveur du message. De cette façon, un acteur peut lancer des messages et collecter les réponses lorsqu'il en a besoin.

Par ailleurs ABCL/1 offre deux modes de transmission des messages : le mode ordinaire, dans lequel les messages sont ajoutés dans la boîte aux lettres du receveur, et le mode express. Lorsqu'un message express est envoyé à un acteur, celui-ci est interrompu pour traiter ce message immédiatement. S'il était déjà en train de traiter un message express, le message reçu est mis dans la boîte aux lettres des messages express, qui est toujours traitée avant la boîte aux lettres des messages ordinaires.

Les différents types de communication comme les modes de transmissions sont destinés à faciliter la programmation avec des acteurs qui reste pourtant assez déroutante. Ainsi les messages express permettent de réaliser des interruptions. Supposons qu'un acteur représente un tableau, et qu'un message permette de trouver un élément dans le tableau. L'acteur décide de

répartir le travail entre plusieurs acteurs qui cherchent dans des parties disjointes du tableau. Dès qu'un acteur a trouvé l'élément cherché, il est inutile pour les autres de poursuivre. L'acteur principal peut alors les interrompre par l'envoi d'un message express. Sans ce moyen, chaque acteur devrait découper sa recherche en étapes élémentaires, en s'envoyant des messages à lui-même afin que le message d'interruption puisse être pris en compte.

Objets et acteurs

On peut se demander dans quelle mesure les langages d'acteurs sont des langages à objets. Un acteur est un objet dans le sens où il détient un état et un comportement, mais, contrairement à un objet, il est actif et s'apparente plutôt à un processus. Cela apparaît clairement dans l'exemple de la factorielle : l'acteur est un objet qui calcule une factorielle, alors qu'un langage à objets classique verrait la factorielle comme un message envoyé à un nombre. Les acteurs sont donc aptes à représenter un comportement ou un calcul, à l'instar des fonctions, ce que les objets sont incapables de faire. Mais les acteurs peuvent aussi représenter un état et des méthodes associées, à l'instar des objets. De tels acteurs peuvent être amenés à créer des acteurs de calcul pour répondre à un message. L'exemple ci-dessous illustre cet aspect.

Il s'agit de représenter le jeu des Tours de Hanoi, et de le résoudre. Nous aurons besoin dans cet exemple de listes de type Lisp. Une liste est notée entre accolades. Nous supposons qu'une liste répond aux messages synchrones *ajoute:*, *car* et *cdr*, et qu'elle peut être parcourue par le message *répéter:* qui prend en argument un bloc avec un argument. Nous dénotons les messages synchrones en utilisant le symbole « ↲ » comme continuation. Les messages synchrones peuvent retourner une valeur en utilisant l'opérateur « ↑ ».

Nous allons utiliser un acteur *Tour* pour représenter une tour, dont l'état contient la pile des disques. Un acteur *Hanoi* représentera l'ensemble du jeu.

```
acteur    Tour
état      pile sommet
filtre
    Empiler: x          [ pile en: sommet mettre: x ↲.
                          sommet ← sommet + 1 ]
    Dépiler             [ sommet ← sommet - 1 ]
    Sommet              [ ↑pile en: sommet ↲ ]
```

L'acteur *Tour* est ici une pile : nous n'avons pas inséré le test qui compare la taille de l'objet empilé avec le sommet courant. Ceci n'est pas important dans cet exemple, car la résolution par programme des Tours de Hanoi assure que les règles du jeu sont respectées. *Sommet* est un message synchrone qui retourne l'objet en sommet de pile. Nous avons supposé que l'acteur qui représente la pile sait répondre aux messages d'accès *en:* et *en:mettre:*. L'empilement utilise un message synchrone.

```
acteur    Hanoi
état      gauche droite centre nd
filtre
    Initialiser
        [ (1 à nd) répéter [ :i | gauche Empiler: (nd - i) ] ]
    déplacer: dép vers: arr
        [ arr Empiler: (dép Sommet ↲).  dép Dépiler ]
    Jouer
        [ jh ← créer JoueHanoi (self).
          jh déplacer: nd de: gauche vers: droite
                par: centre tag: 1 → jh ]
```

L'acteur *Hanoi* représente les Tours de Hanoi. *Initialiser* permet de mettre *nd* disques de tailles décroissantes sur la tour de gauche. *déplacer:vers:* déplace un disque de la tour passée en premier argument vers celle passée en second argument ; il envoie le message synchrone *Sommet* à la tour de départ, empile la valeur retournée sur la pile d'arrivée, et dépile la tour de départ. *Jouer* permet de lancer la résolution du jeu. *Jouer* crée un acteur *JoueHanoi* chargé de résoudre le jeu. Rappelons l'algorithme séquentiel récursif qui résout les Tours de Hanoi :

```
-- déplacer n disques de la tour dép vers la tour arr
-- en utilisant la tour intermédiaire inter
procédure Hanoi (n : entier; dép, arr, par : tour) {
    si n ≠ 0 alors {
        Hanoi (n-1, dép, inter, arr);
        DéplaceDisque (dép, arr);
        Hanoi (n-1, inter, arr, dép);
    }
}
```

Avec les acteurs, la difficulté provient de la résolution parallèle qui doit néanmoins produire une liste *ordonnée* de déplacements de disques. Selon la technique utilisée pour calculer la factorielle, et à l'aide d'une jointure de continuations, nous allons créer des acteurs qui représentent les étapes du calcul, c'est-à-dire les sous-séquences de déplacements de disques.

```
acteur    JoueHanoi
état      hanoi
filtre
    déplacer: 1 de: D vers: A par: M tag: t → cont
        [ cont tag: t liste: {D A} ]
    déplacer: n de: D vers: A par: M tag: t → cont
        [ c ← créer Jointure (cont, t, {D A}, 0).
          self déplacer: n-1 de: D vers: M par: A tag: t*2 → c.
          self déplacer: n-1 de: M vers: A par: D tag: t*2+1 → c.
        ]
    tag: t liste: listedépl
        [ listedépl répéter:
            [ :d | hanoi déplacer: (d car ↲) vers: (d cdr ↲) ↲]
        ]
```

Lorsqu'il reçoit le message de résolution *déplacer:de:vers: par:tag:*, l'acteur *JoueHanoi* crée un acteur *Jointure* pour la jointure des continuations et s'envoie deux messages de résolution pour les deux sous-problèmes. L'acteur de jointure est initialisé avec le déplacement médian et la continuation de *JoueHanoi*. Il attend de recevoir les deux listes de déplacements,

les combine avec le déplacement médian, et envoie le résultat à la continuation. Voyons maintenant l'acteur de jointure :

```
acteur    Jointure
état      cont t listedépl attente
filtre
    tag: t*2 liste: l
          [ listedépl ← l ajoute: listedépl ↲. self envoyer ]
    tag: t*2 +1 liste: l
          [ listedépl ← listedépl ajoute: l ↲. self envoyer ]
    envoyer
          [ attente ← attente+1.
           (attente = 2) siVrai: [ cont tag: t liste: listedépl ] ]
```

Afin de combiner les listes de déplacements correctement, l'acteur de jointure doit pouvoir distinguer les deux listes qu'il reçoit. Pour cela, les messages de résolution transportent une étiquette, appelée *tag*, qui numérote chaque résolution de façon unique : la résolution d'étiquette *n* déclenche deux résolutions d'étiquettes *2*n* et *2*n+1*. L'acteur de jointure est initialisé avec l'étiquette de la résolution pour laquelle il a été créé ; il peut donc déterminer l'origine des listes qu'il reçoit et combiner les déplacements en conséquence. Le message *envoyer* permet d'envoyer le déplacement final à la continuation, lorsque les deux sous-listes ont été reçues.

Le message *tag:liste:* est envoyé par *JoueHanoi* lorsqu'il a un seul disque à déplacer, et par *Jointure* lorsqu'il a combiné les listes avec le déplacement médian. Il est reçu soit par *Jointure*, auquel cas il contient les sous-listes de déplacements, soit par *JoueHanoi* lui-même lorsque la résolution est terminée. Dans ce cas, la valeur de l'étiquette n'est plus utile. Lorsque *JoueHanoi* reçoit la résolution finale, il énumère la liste des déplacements et envoie à *Hanoi* des messages de déplacement de disque *déplace:vers:*. Ces envois de messages sont synchrones pour assurer leur traitement dans l'ordre d'émission ; sinon, tout le travail précédent aurait été inutile.

L'exemple ci-dessous initialise un acteur *Hanoi* avec deux disques et lance une résolution, qui est illustrée figure 28.

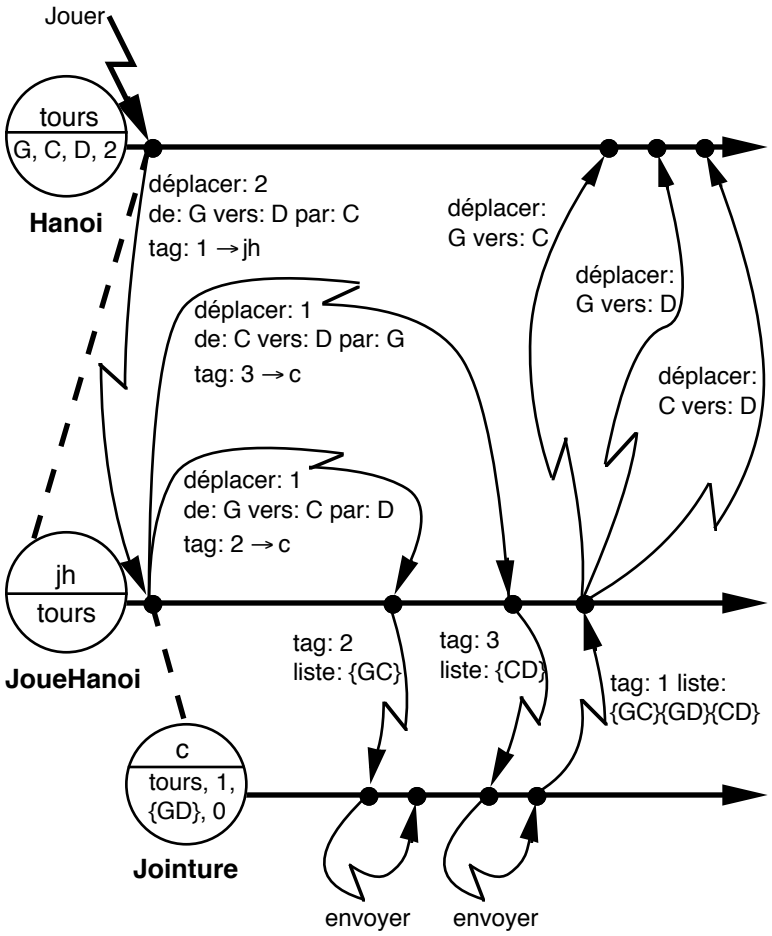

Figure 28 - Résolution des Tours de Hanoi

```
tours ← créer Hanoi (
                créer Tour (créer Tableau, 0),   -- gauche
                créer Tour (créer Tableau, 0),   -- centre
                créer Tour (créer Tableau, 0),   -- droite
                2).                              -- nd
tours Initialiser. tours Jouer.
```

Le degré de parallélisme obtenu est assez faible : l'acteur de jointure fonctionne en parallèle avec l'acteur *JoueHanoi*, mais c'est ce dernier qui fait l'essentiel du travail. Si l'on a *n* disques, il y a création d'un seul acteur *JoueHanoi*, et de $2^{n-1}-1$ acteurs de jointure. Le temps de résolution est exponentiel en fonction de *n*, car l'acteur *JoueHanoi* s'envoie 2^{n-1} messages qu'il traite séquentiellement.

On pourrait augmenter le parallélisme en créant un acteur *JoueHanoi* à chaque décomposition du problème. Pour *n* disques, il y aurait création de *2*(n-1)* acteurs JoueHanoi et de $2^{n-1}-1$ acteurs de jointure, et le temps de résolution serait linéaire en fonction de *n*.

Conclusion

La programmation avec un langage d'acteurs n'est pas simple, mais cela est vrai de tous les langages parallèles. Par contre, le modèle des acteurs est facile à comprendre, ce qui n'est pas le cas de tous les modèles du parallélisme.

Les langages d'acteurs sont plus proches des langages de prototypes que des langages de classes : ils utilisent la délégation, et la création d'acteurs est proche du clonage. Cette ressemblance a des raisons historiques : la plupart des langages d'acteurs ont été développés au-dessus de Lisp, et ils sont contemporains des premiers langages de prototypes. Il en résulte que les langages d'acteurs ne sont pas typés, qu'ils n'ont pas de mécanismes de modularité, et qu'ils emploient la liaison dynamique. Les travaux sur les langages d'acteurs se sont focalisés essentiellement sur la sémantique de l'envoi de message, au détriment de ces autres aspects.

Les langages d'acteurs et les langages de prototypes explorent des directions indépendantes qui se démarquent du modèle strict des langages de classe. Ils permettent aussi de mieux comprendre l'essence de la programmation par objets, et l'on peut s'attendre à des retombées de ces travaux sur les langages à objets plus classiques.