Réflexivité et programmation du jeu de Go

Tristan Cazenave
Laboratoire d'Intelligence Artificielle
Université Paris 8, Saint Denis, France
cazenave@ai.univ-paris8.fr

Résumé

La réflexivité peut intervenir de multiples manières dans un programme de Go. Je présente ici trois aspects de la réflexivité dans la programmation du jeu de Go. Le premier aspect concerne l'auto-observation d'un programme de Go pour engendrer de nouvelles connaissances lui permettant de trouver plus rapidement des solutions aux problèmes qu'il doit résoudre. Le second aspect est l'utilisation d'un algorithme de recherche par ce même algorithme de recherche pour sélectionner les coups à envisager dans l'algorithme de recherche, ce qui lui permet d'être beaucoup plus efficace que l'algorithme simplement récursif habituellement utilisé. Le troisième aspect concerne l'utilisation d'un méta-niveau dans un programme de Go pour départager les recherches que l'on a intérêt à continuer de celles qui sont devenues inutiles.

1 Introduction

Une définition restrictive de la réflexivité, défendue par certains chercheurs en Intelligence Artificielle, est l'application d'un système à lui-même [Pitrat, 1990]. On peut ainsi utiliser un système qui découvre des connaissances pour découvrir des connaissances de découverte de connaissances [Lenat, 1983; Cazenave, 1999], ou utiliser un système qui contrôle l'utilisation des connaissances pour contrôler l'utilisation des connaissances de contrôle. Nous appellerons cette vision de la réflexivité, la réflexivité forte.

Une autre définition de la réflexivité couramment utilisée dans la communauté des chercheurs en langage de programmation [Tanter *et al.*, 2003] consiste pour un programme à avoir accès, lors de son exécution, à son propre état et à ses structures de haut niveau, et à pouvoir les modifier. On retrouve cette forme de réflexivité dans des langages de programmation comme Lisp, Java, Smalltalk, .NET, Ruby ou Python.

Dans cet article nous nous intéressons à la réflexivité comme la capacité que peut avoir un programme d'observer son propre comportement, et aux bénéfices que peut apporter cette auto-observation pour rendre son fonctionnement plus efficace.

La réflexivité peut être utilisée dans un système complexe aussi bien pour permettre au système d'apprendre à trouver des solutions plus rapidement, que pour lui permettre de comprendre les raisons de ses succès ou de ses échecs et lui donner ainsi des indices sur les nouveaux essais à tenter, ou pour lui permettre de mieux contrôler son raisonnement en arrêtant avant qu'ils ne soient terminés les raisonnements qui sont inutiles ou qui ont une utilité trop faible.

Cette forme de réflexivité est liée au métaraisonnement dans les systèmes temps réél complexes composés de nombreux modules interdépendants. Le contrôle des ressources allouées à des algorithmes de recherche heuristique est par exemple un problème que l'on rencontre aussi bien dans les jeux [Russell and Wefald, 1989] que dans d'autres systèmes complexes [Pitrat, 1990; Russell and Wefald, 1991; Horvitz, 2001; Mouaddib *et al.*, 2002]. Ce type de problème était déjà abordé dans un des premiers démonstrateurs automatique de théorèmes [Pitrat, 1966] qui démontrait des meta-théorèmes de façon à améliorer son fonctionnement. C'est un thème récurrent et fondamental en Intelligence Artificielle [Ginsberg and Geddis, 1991; Russell, 1997; Minton, 1990; 1996].

L'article est structuré en trois sections : l'apprentissage par auto-observation, les recherches sélectionnant les coups à l'aide de recherches, les métaraisonnements sur l'utilité des recherches.

2 L'apprentissage par auto-observation

L'apprentissage par auto-observation [Cazenave, 1996b] consiste à mémoriser dans une trace les raisonnements d'un programme de résolution de problème pour ensuite analyser cette trace afin de créer de nouvelles connaissances qui permettront au programme de résoudre des problèmes plus rapidement. Ce type d'apprentissage a montré son utilité en programmation des jeux et d'une façon plus générale en résolution de problèmes.

2.1 Résolution de problème

Au départ un système d'apprentissage par auto-observation dispose d'une définition des règles du jeu, d'une définition des buts qu'il doit atteindre, et de métaconnaissances qui lui permettent de transformer les connaissances qu'il engendre.

La première étape de l'apprentissage est la résolution de problèmes. Lorsqu'il est confronté à de nouveaux problèmes, il essaie des coups et vérifie si un but est atteint après ces coups. Lorsqu'un coup atteint un but alors qu'aucune règle ne lui permet jusqu'alors de déduire que ce coup atteint le but, le système déclenche alors le processus d'apprentissage qui lui permettra de créer une règle qui trouvera ce coup. Pour cela il a besoin de la trace des raisonnements qui lui ont permis de trouver que le coup atteignait le but. Le système mémorise donc toutes les règles qu'il a déclenché lors de la résolution de problème. Ce qui constitue la trace du raisonnement.

2.2 Analyse de la trace

Lorsque le système a détecté qu'il pouvait apprendre une nouvelle connaissance, il remonte dans la trace pour trouver les faits décrivant la position avant que le coup, qui sont à l'origine de la vérification du but après le coup. Une fois ces faits regroupés, il les assemble en une règle qui permettra de déduire plus rapidement le coup à jouer. Cette règle qui est au départ un ensemble de faits, est généralisée en transformant les variables instanciées en variable, puis ses conditions sont réordonnées de façon à être vérifiées efficacement [Cazenave, 1996a; 1998b].

2.3 Métaprogrammation

Les règles ainsi apprises sont transformées par un métaprogramme qui trouve tous les coups qui sont susceptibles d'empécher d'atteindre un but à partir d'une règle déduisant un coup atteignant ce but [Cazenave, 1998a]. On appelle cet ensemble de coups des coups forcés (ce sont les seuls coups à envisager pour empêcher d'atteindre le but). Les règles sur les coups forcés sont utilisées pour engendrer des règles sur les situations ou un but est atteint. Les règles sur les buts atteints sont elles mêmes utilisées pour apprendre les règles sur les coups qui atteignent un but, et ainsi de suite.

Pour éviter au système d'engendrer trop de règles, des métarègles de sélection des règles engendrées sont utilisées. On peut par exemple dans le cas du jeu de Go, ne retenir que les règles sur les coups forcés qui concluent sur un petit nombre de coups forcés.

Il est possible de remplacer le programme d'apprentissage par auto-observation par un programme qui engendre directement les règles par métaprogrammation. On peut aussi engendrer par métaprogrammation certains des métaprogrammes nécessaires à la génération des programmes [Cazenave, 1999], ce qui est une forme de réflexivité forte.

3 Les recherches sélectionnant les coups à l'aide de recherches

L'algorithme qui a en général le plus de succès dans les jeux de réflexion est l'Alpha-Béta. Toutefois, dans la plupart des jeux classiques, le nombre de coups légaux est relativement petit (de l'ordre d'une trentaine aux Echecs par exemple) ce qui permet à l'Alpha-Béta de faire des recherches relativement profondes (de l'ordre de 12 demi-coups actuellement) et donc d'avoir des programmes de bon niveau. Pour le jeu de Go, la situation est bien différente des autres jeux classiques, puisque le nombre de coups possibles est de l'ordre de 250 et que les recherches sont en général plus profondes que dans les autres jeux.

On est donc amené au jeu de Go à utiliser des algorithmes sélectifs de recherche. Le problème est alors de sélectionner un petit nombre de coups sans toutefois oublier les coups importants. L'apprentissage par auto-observation permet d'apprendre des règles qui sélectionnent surement un petit nombre de coups. Une autre manière de sélectionner les coups est d'engendrer dynamiquement à chaque fois les explications sur la réussite d'un but, de façon à en tirer les coups forcés qui permettront d'empêcher l'adversaire d'atteindre ce but.

Les menaces généralisées [Cazenave, 2002] sont basées sur ce principe. A chaque noeuds Min de la recherche, le programme essaie des coups Max au lieu des coups Min. Puis il relance une recherche après chaque coup Max. Si un de ces coups Max atteint le but, il retient dans une structure de données les raisons qui font que le coup atteint le but. Puis il en déduit un petit nombre de coups Min qui sont les seuls susceptibles d'empêcher Max d'atteindre le but.

Le pseudo code pour la vérification des menaces est le suivant :

```
MenaceMin (menace, ordre, raisons) {
  Si butAtteint ()
    return 1;
  res = 0;
  mg = menaceGauche (menace, ordre);
  if (MenaceMax (mg, ordre, raisons) == 1) {
    res = 1;
    coups = trouveCoupsMin (ordre, raisons);
    md = menaceDroite (menace, mg);
    o = ordreMenace (md);
    for (c dans coups et tant que res == 1) {
      joue (c, MinColor);
      if (MenaceMax (md, o, raisons) == 0)
        res = 0;
      dejoueCoup ();
    }
  }
  return res;
}

MenaceMax (menace, ordre, raisons) {
  res = 0;
  coups = trouveCoupsMax (ordre);
  oteCoupOrdre (menace, ordre);
  for (c dans coups et tant que res == 0) {
```

```
    joue (c, MaxColor);
    initialise (r);
    if (MenaceMin (menace, ordre-1, r) == 1) {
      res = 1;
      ajouteRaisonsCoup (c, r);
      raisons = r;
    }
    dejoueCoup ();
  }
  return res;
}
```

L'ordre d'une menace est le nombre maximum de coups de suite de la couleur Max qu'on a le droit de jouer en vérifiant la menace.

Lorsqu'on vérifie une menace, pour un noeud Min, il y a deux sous-arbres, le sous-arbre gauche et le droit. Par exemple dans la figure 1, dans le noeud qui est sous la racine de la menace (6,3,2,0), le sous-arbre gauche est une menace (2,1,0) alors que le sous-arbre droit est une menace (4,2,1,0).

L'algorithme de recherche mémorise dynamiquement les raisons pour lesquelles ce même algorithme de recherche avec un ordre inférieur a trouvé la solution à un problème. Ces raisons qui constituent une trace de l'exécution de l'algorithme de recherche sont utilisées pour trouver les coups de Min (et constituent un petit sous ensemble de l'ensemble des coups possibles).

Cet algorithme de recherche est non seulement beaucoup plus rapide qu'un algorithme Alpha-Béta naïf qui essaierait tous les coups possibles, mais il est aussi plus rapide et plus sûr qu'un algorithme Alpha-Béta sophistiqué qui n'envisage que certains coups en se basant sur des connaissances du jeu de Go. Ainsi pour la résolution du jeu d'Atarigo 6x6, l'algorithme de recherche utilisant la menace généralisée (6,3,2,0) est 270 fois plus rapide qu'un Alpha-Béta utilisant les mêmes optimisations de recherche [Cazenave, 2002] (tables de transposition, deux coups qui tuent, heuristique de l'historique).

La figure 1 donne des exemples d'arbres correspondant à des menaces généralisées. Dans ces arbres toutes les feuilles correspondent à des positions où le but est atteint pour le joueur Max. Chaque branche de l'arbre vers la gauche correspond à un coup gagnant pour Max. Chaque branche vers la droite correspond à l'ensemble de coups Min qui sont susceptibles d'empêcher Max d'exécuter la menace associée à la branche vers la gauche qui part du même noeud. On peut compter pour chaque arbre le nombre de coups de chaque ordre pour Max. Ainsi l'abre le plus simple correspondant à un coup gagnnant pour Max ne contient qu'un seul coup d'ordre un. On notera cet arbre (1,0). Le premier nombre dans cette notation correspond au nombre de coups d'ordre un dans l'arbre, le deuxième nombre au nombre de coups d'ordre deux, et ainsi

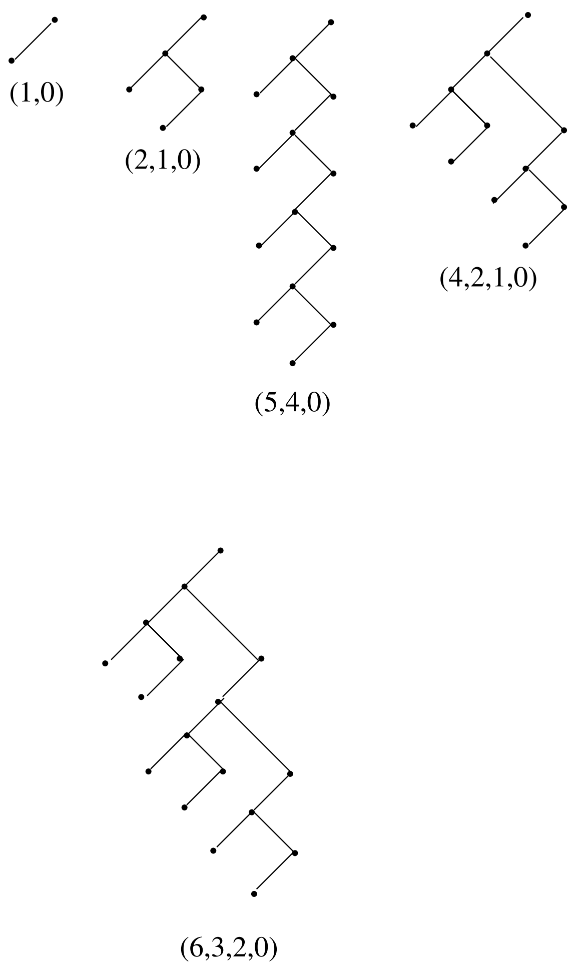

Figure 1: Arbres de menaces généralisées.

de suite jusqu'au premier zéro qui signifie qu'il n'y a pas de coups d'ordre supérieur à l'ordre correspondant au zéro.

Pour tous ces arbres, quand on est à un noeud où il y a une branche vers la droite et une branche vers la gauche, on a toujours le sous-arbre gauche inclus dans le sous-arbre droit. Cette contrainte sur la forme des arbres vient de ce qu'il est la plupart du temps plus difficile d'atteindre le but pour Max après un coup de Min qu'avant.

Pour les menaces généralisées complexes comme (6,3,2,0), plusieurs arbres correspondent à la même suite de nombres.

4 Les métaraisonnements sur l'utilité des recherches

Dans un programme de jeu de Go, l'efficacité du métaraisonnement dépend des résultats des recherches heuristiques, et la décision de continuer ou non les recherches heuristiques dépend du métaraisonnement et de raisonnement stratégiques sur des objets plus complexes que ceux traités par les recherches

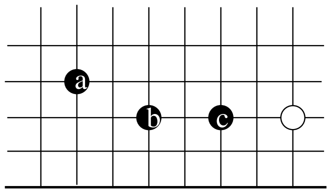

Figure 2: Trois calculs de connexion possibles.

heuristiques. Le méta niveau et le niveau de base sont interdépendants. Nous expliquons quelques unes de ces dépendances et la façon dont nous les traitons.

Un programme de Go effectue un grand nombre de recherches tactiques dont les résultats lui permettent de construire des objets complexes comme les groupes. Une approche basique consiste à faire toutes les recherches tactiques possibles, puis à construire les objets complexes une fois toutes les recherches effectuées. Une approche plus fine est d'entrelacer les recherches tactiques avec la construction des objets complexes. En effet, les objets complexes, même partiellement construits nous renseignent sur l'utilité des recherches tactiques restantes. En entrelaçant recherches tactiques et contruction d'objets complexes, on peut ainsi décider d'abandonner des recherches couteuses et inutiles avant de les avoir terminées.

4.1 Construction des groupes

Nous présentons ici un cas particulier qui est la construction des groupes à partir des résultats des recherches sur la connexion de deux chaînes.

La figure 2 donne un exemple où le programme trouve trois connexions à calculer. Les connexions à calculer sont les connexions de a avec b, de b avec c, et de a avec c.

Les groupes sont construits à partir des calculs de connexion. Si deux chaînes ne peuvent être déconnectées, comme c'est par exemple la cas pour les pierres b et c dans la figure 2, elles font partie du même groupe.

Le programme doit toutefois faire attention à la non transitivité des connexions [Cazenave and Helmstetter, 2004]. Dans le cas de la figure 2, a est connecté à b, b est connecté à c, et a est connecté à c. Il y a transitivité de la connexion, toutefois ce n'est pas toujours le cas.

Le programme qui détecte si les connexions complexes sont transitives est plus efficace que le programme qui calcule directement les connexions composées. Par exemple, toujours dans la figure 2, il est plus efficace de calculer que a est connecté à b et que b est connecté à c, plutôt que de calculer si a est connecté à c. On comprend la raison de cette efficacité accrue en raisonnant sur le coût de chaque recherche. Si le facteur de branchement est b pour une connexion (et donc 2b pour une connexion composée), et que la profondeur de recherche pour prouver une connexion est p (et donc 2p pour une connexion

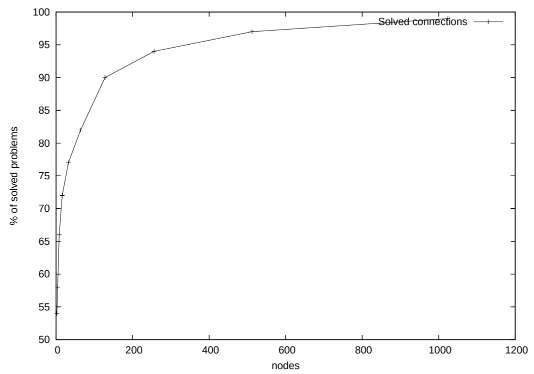

Figure 3: Profil pour les problèmes de connexion

composée). Chercher que a et connecté à b et que b est connecté à c coutera $2b^p$ alors que le calcul de la connexion de a à c coutera $(2b)^{2p}$. Le calcul d'une connexion composée est beaucoup plus couteux que le calcul de chacune des connexions indépendamment. Les calculs de connexion sont indépendants lorsque les traces (les raisons des menaces généralisées) sont disjointes.

On construit les groupes en ajoutant au groupe les chaînes qui ne peuvent pas être déconnectées d'une chaîne appartenant déjà au groupe, tout en vérifiant qu'il y a bien transitivité de la connexion avec les autres chaînes du groupe.

4.2 Une implémentation simple

Une implémentation simple d'un programme de Go consiste à effectuer tous les calculs tactiques les uns après les autres pour ensuite construire les groupes et utiliser les résultats des recherches tactiques dans des objets plus complexes.

4.3 Interactions entre la construction des groupes et les recherches

Une implémentation plus fine consiste à entrelacer construction des groupes et recherches tactiques. Dans le cas des recherches de connexions, on peut fixer une suite de seuils pour les recherches. A chaque fois que toutes les recherches ont atteint un certain seuil, on déclenche le mécanisme qui permet d'arrêter les recherches inutiles. Par exemple dans le cas de la figure 2, dès que les recherches ont trouvé que a est connecté à b et que b est connecté à c, et que la connexion est transitive, on arrête la recherche sur la connexion entre a et c.

Les figures 3 et 4 donnent le pourcentage de problèmes résolus en fonction du nombre de noeuds cherchés pour les problèmes de connexion et de déconnexion. On voit bien que la progression est logarithmique. Il est donc probablement préferable de fixer une suite de seuils pour les recherches tactiques avec une progression exponentielle.

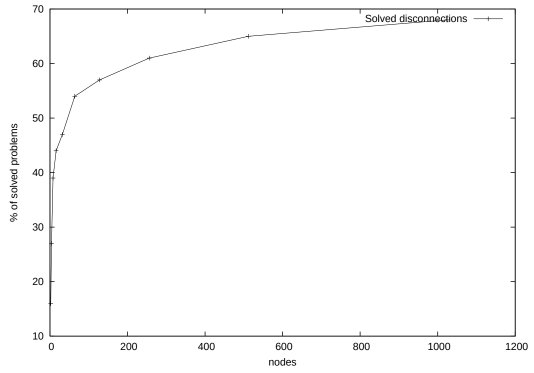

Figure 4: Profil pour les problèmes de déconnexion

4.4 Autres interactions possibles

De même qu'on peut détecter que des calculs de connexions complexes sont inutiles parce que déductible de manière plus simple, on peut aussi détecter l'inutilité d'autres calculs. Par exemple si on a trouvé deux yeux indépendants dans un même groupe, il devient inutile de faire une recherche sur la vie de ce groupe.

4.5 Ordonnancement heuristique des recherches

Une autre heuristique possible est d'ordonner les recherches par complexités présumées, en partant de la plus simple vers la plus complexe. Une bonne heuristique pour évaluer la complexité de la recherche sur une connexion est de compter le nombre de coups de suite qu'il faut jouer pour connecter les deux chaînes, c'est à dire l'ordre de la connexion au sens des menaces généralisées. Le programme peut ainsi continuer à éviter des recherches complexes inutiles puisqu'il disposera avant de les commencer des combinaisons de recherches plus simples. Par exemple, les problèmes de la figure 2 sont respectivement d'ordres 2, 1 et 3. Le programme commencera par les recherches d'ordre 1 et 2, et évitera la recherche d'ordre 3 car elle peut être déduite de la combinaison des deux recherches plus simples.

5 Conclusion

J'ai montré trois utilisations de la capacité qu'a un programme d'observer une partie de son propre fonctionnement en rapport avec la programmation du Go. Dans ces trois cas, l'utilisation de l'observation du programme par lui même lui permet de gagner du temps sur une implémentation n'utilisant pas cette auto-observation. Dans le cas de l'apprentissage par auto-observation, le programme crée de nouvelles connaissances qui lui permettent de résoudre les problèmes plus rapidement. Dans le cas des menaces généralisées, l'observation de la trace de ses calculs lui permet d'être beaucoup plus efficace que les algorithmes plus classiques. Dans le cas des métaraisonnempents sur l'utilité des recherches, le programme évite de perdre du temps dans des recherches inutiles en alternant les recherches avec la détection des recherches devenues inutiles.

References

[Cazenave and Helmstetter, 2004] T. Cazenave and B. Helmstetter. Search for transitive connections. *Information Sciences*, 2004.

[Cazenave, 1996a] T. Cazenave. Automatic ordering of predicates by metarules. In *5th International Workshop on Metaprogramming and Metareasoning in Logic*, Bonn, 1996.

[Cazenave, 1996b] T. Cazenave. Système d'Apprentissage Par Auto-Observation. Application au jeu de Go. Phd thesis, Université Paris 6, December 1996.

[Cazenave, 1998a] T. Cazenave. Metaprogramming Forced Moves. In *ECAI 1998*, pages 645–649, Brighton, UK, 1998.

[Cazenave, 1998b] T. Cazenave. Speedup Mechanisms For Large Learning Systems. In *IPMU 1998*, Paris, France, 1998.

[Cazenave, 1999] T. Cazenave. Metaprogramming domain specific metaprograms. In *Reflection 99*, volume 1616 of *Lecture Notes in Computer Science*, pages 235–249. Springer, 1999.

[Cazenave, 2002] T. Cazenave. A Generalized Threats Search Algorithm. In *Computers and Games 2002*, volume 2883 of *Lecture Notes in Computer Science*, pages 75–87, Edmonton, Canada, 2002. Springer.

[Ginsberg and Geddis, 1991] M. L. Ginsberg and D. F. Geddis. Is there any need for domain-dependent control information? In *AAAI-91*, pages 452–457, 1991.

[Horvitz, 2001] E. Horvitz. Principles and applications of continual computation. *Artificial Intelligence*, 126(1-2):159–196, 2001.

[Lenat, 1983] D. B. Lenat. Eurisko: A program that learns new heuristics and domain concepts. *Artificial Intelligence*, 21:61–98, 1983.

[Minton, 1990] S. Minton. Quantitative Results Concerning the Utility of Explanarion Based Learning. *Artificial Intelligence*, 42(2-3):363–392, 1990.

[Minton, 1996] S. Minton. Is there any need for domain-dependent control information : A reply. In *AAAI-96*, 1996.

[Mouaddib *et al.*, 2002] A.-I. Mouaddib, E. Grégoire, and J.-F. Dauchez. An intelligent system combining different resource-bounded reasoning techniques. *Applied Intelligence*, 17(2):127–140, 2002.

[Pitrat, 1966] J. Pitrat. Réalisation de programmes de démonstration de théorèmes utilisant des méthodes heuristiques. Thesis, Université de Paris, 1966.

[Pitrat, 1990] J. Pitrat. *Métaconnaissance - Futur de l'Intelligence Artificielle*. Hermès, Paris, 1990.

[Russell and Wefald, 1989] S. J. Russell and E. Wefald. On optimal game tree search using rational meta-reasoning. In *IJCAI 89*, pages 334–340. Morgan Kaufmann, Detroit, Michigan, August 1989.

[Russell and Wefald, 1991] S. J. Russell and E. Wefald. Principles of metareasoning. *Artificial Intelligence*, 49:361–395, 1991.

[Russell, 1997] S. J. Russell. Rationality and intelligence. *Artificial Intelligence*, 94(1-2):57–77, 1997.

[Tanter *et al.*, 2003] E. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *OOPSLA 2003*, pages 27–46, 2003.