Étude sémantique d'un langage intermédiaire de type Static Single Assignment

Boris Yakobowski

INRIA Rocquencourt
DEA Programmation

13 Septembre 2004

Plan

Plan

Projet(s) d'accueil

Projet Cristal

Travaille à la conception, à l'implémentation et à l'établissement des fondements théoriques des langages de programmation fortement typés.

ARC Concert

Cherche à déterminer s'il est faisable de réaliser un *compilateur optimisant* qui soit *certifié* (avec une preuve Coq d'équivalence sémantique entre le code source et le code machine engendré).

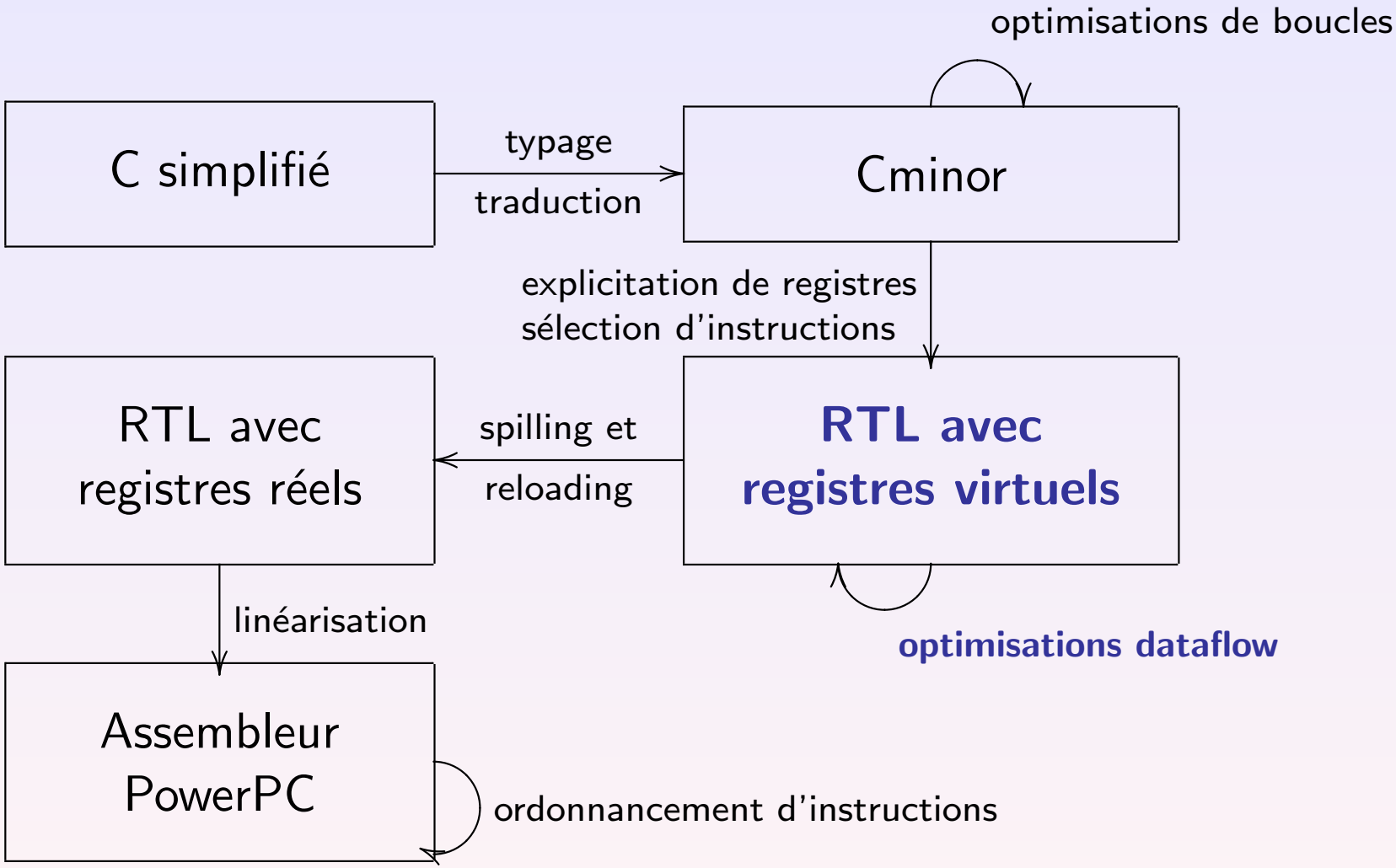
Projet(s) d'accueil

Projet Cristal

Travaille à la conception, à l'implémentation et à l'établissement des fondements théoriques des langages de programmation fortement typés.

ARC Concert

Cherche à déterminer s'il est faisable de réaliser un *compilateur optimisant* qui soit *certifié* (avec une preuve Coq d'équivalence sémantique entre le code source et le code machine engendré).


optimisations de boucles
C simplifié
typage
traduction
Cminor
explicitation de registres
sélection d'instructions
RTL avec
registres réels
spilling et
reloading
RTL avec
registres virtuels
optimisations dataflow
linéarisation
Assembleur
PowerPC
ordonnancement d'instructions

Problèmes dans l'implémentation actuelle

Performances insuffisantes

Les optimisations actuellement effectuées sont à base de *data-flow*, et sont peu efficaces algorithmiquement.

Forme SSA

En forme SSA, toutes les optimisations actuellement effectuées dans Concert peuvent être faites en un temps linéaire.

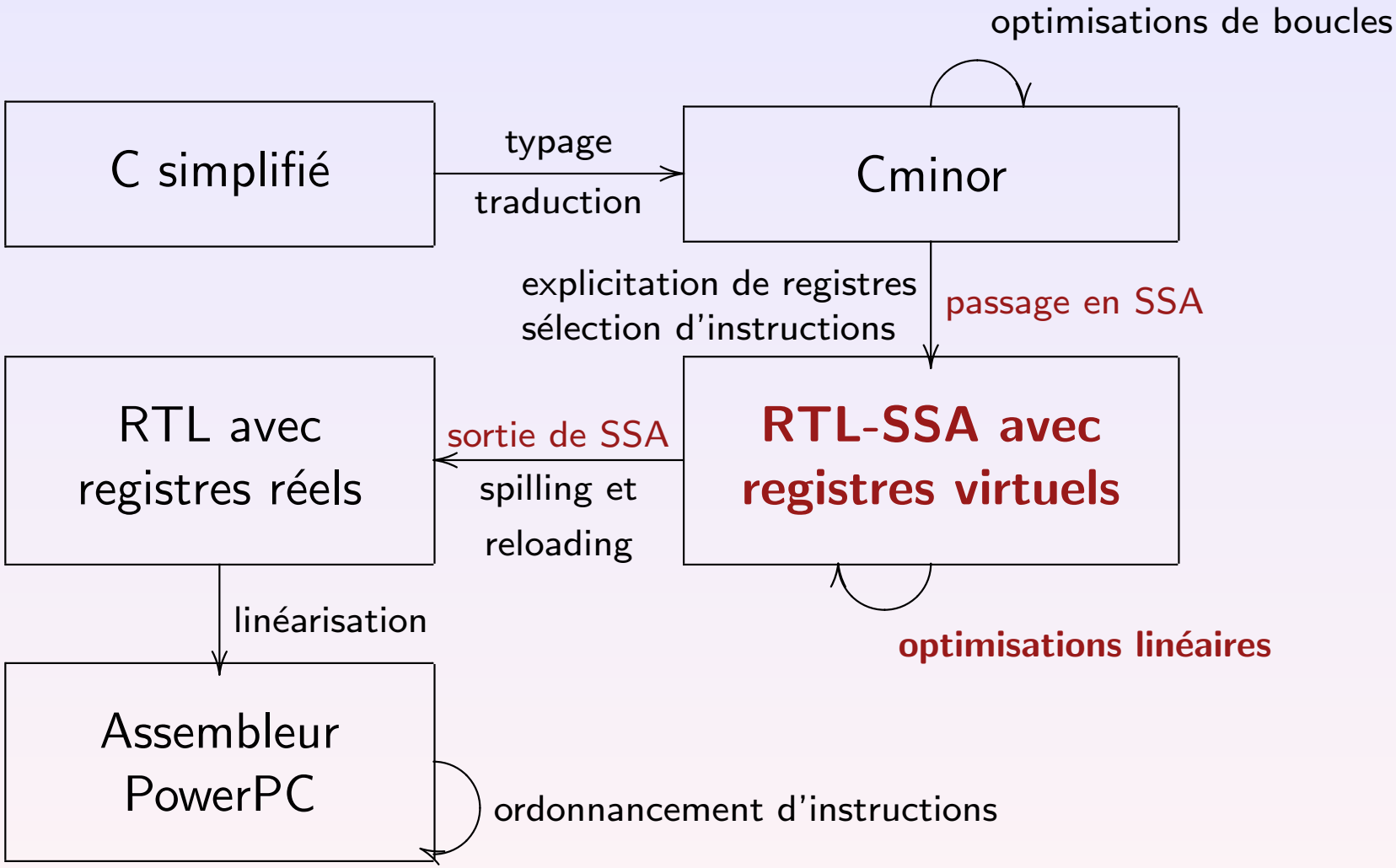
Problèmes dans l'implémentation actuelle

Performances insuffisantes

Les optimisations actuellement effectuées sont à base de *data-flow*, et sont peu efficaces algorithmiquement.

Forme SSA

En forme SSA, toutes les optimisations actuellement effectuées dans Concert peuvent être faites en un temps linéaire.


optimisations de boucles
C simplifié
typage
traduction
Cminor
explicitation de registres
sélection d'instructions
passage en SSA
RTL avec
registres réels
sortie de SSA
spilling et
reloading
RTL-SSA avec
registres virtuels
optimisations linéaires
linéarisation
Assembleur
PowerPC
ordonnancement d'instructions

La forme SSA est mal comprise

- Pas de sémantique formelle
- Les propriétés devant être vérifiées par une fonction en forme SSA sont rarement énoncées
- Conventions difficiles à vérifier (toutes les variables doivent êtres initialisées au début du programme)
- Classe des optimisations licites non explicitée

Apports de ce stage

1. Algorithmes de passage et de sortie de la forme SSA prouvés
2. Large classe de fonctions transformables
3. Formulation explicite des propriétés qu'une fonction en forme SSA doit vérifier
4. Preuve de la plupart des optimisations utilisées dans Concert

Plan

Propriétés d'une fonction en forme SSA

En forme SSA, les propriétés suivantes doivent être vérifiées :

Definition (Unicité de la définition)

Chaque registre ne peut apparaître que dans un seul membre gauche d'une affectation (variables assignées statiquement une unique fois).

Definition (Domination)

Tout registre lu dans une instruction doit être dominé par sa définition.

Propriétés d'une fonction en forme SSA

En forme SSA, les propriétés suivantes doivent être vérifiées :

Definition (Unicité de la définition)

Chaque registre ne peut apparaître que dans un seul membre gauche d'une affectation (variables assignées statiquement une unique fois).

Definition (Domination)

Tout registre lu dans une instruction doit être dominé par sa définition.

Renommage des variables

Example

Facile pour les blocs linéaires, il suffit de renommer toutes les variables :

$$
\begin{array}{ll}
x = 2; & x_1 = 2; \\
y = 3; & y_1 = 3; \\
z = x + y; & z_1 = x_1 + y_1; \\
x = 4; & x_2 = 4; \\
z = z + x; & z_2 = z_1 + x_2 \\
\mathrm{return}(z); & \mathrm{return}(z_2);
\end{array}
$$

Renommage des variables

Example

Facile pour les blocs linéaires, il suffit de renommer toutes les variables :

$$\begin{array}{ll} x = 2; & x_1 = 2; \\ y = 3; & y_1 = 3; \\ z = x + y; & z_1 = x_1 + y_1; \\ x = 4; & x_2 = 4; \\ z = z + x; & z_2 = z_1 + x_2 \\ \mathrm{return}(z); & \mathrm{return}(z_2); \end{array}$$

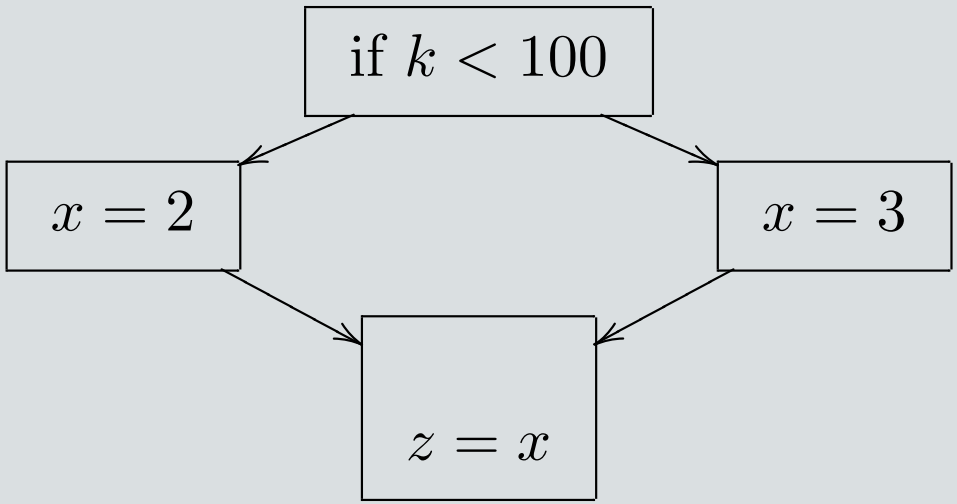
SSA et flot de contrôle

Fonctions ϕ

Un problème se pose lorsque plusieurs chemins d'exécutions se rejoignent en un même bloc (par exemple après un *if*).

On insère alors des instructions virtuelles ϕ, qui indiquent de quel endroit on arrive.

Example



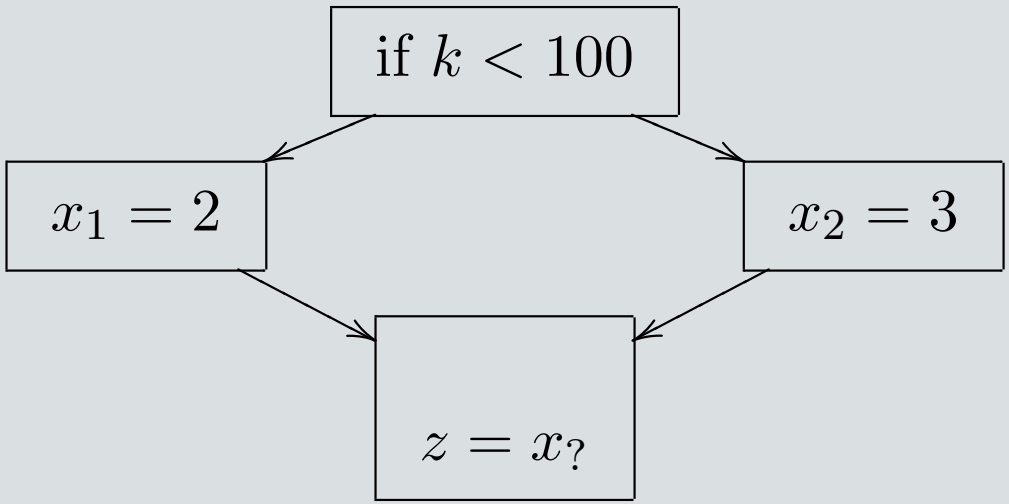
SSA et flot de contrôle

Fonctions ϕ

Un problème se pose lorsque plusieurs chemins d'exécutions se rejoignent en un même bloc (par exemple après un *if*).

On insère alors des instructions virtuelles ϕ, qui indiquent de quel endroit on arrive.

Example

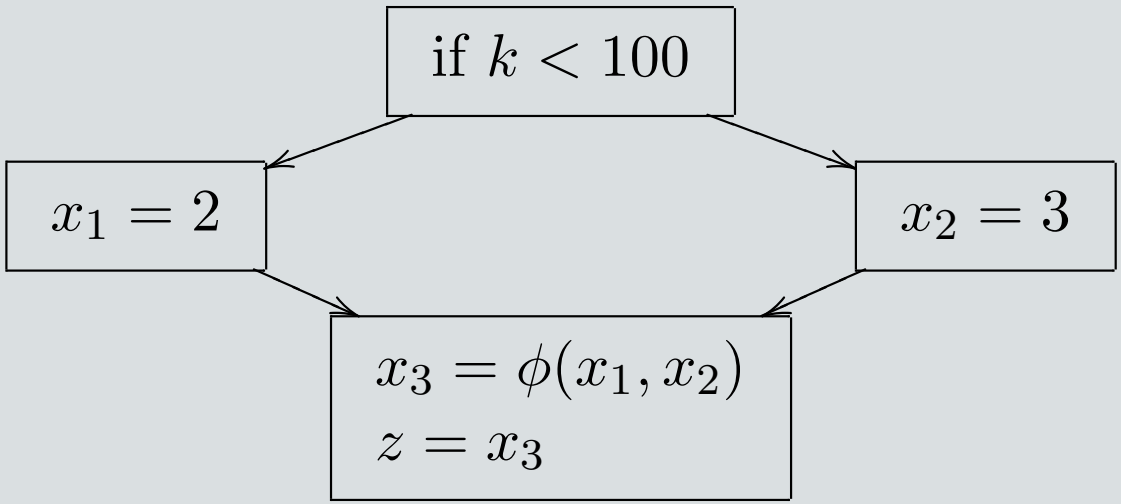


SSA et flot de contrôle

Fonctions ϕ

Un problème se pose lorsque plusieurs chemins d'exécutions se rejoignent en un même bloc (par exemple après un *if*).
On insère alors des instructions virtuelles ϕ, qui indiquent de quel endroit on arrive.

Example



Où insérer des ϕ ?

Question : Combien de ϕ faut-il insérer (et où ?)

Réponse traditionnelle : en chaque point du flot de contrôle où deux assignation différentes se rencontrent (pour la 1ère fois) (sur la frontière de domination) :
Forme SSA *minimale*

Problèmes

- Obtenir la frontière de domination n'est pas trivial : problème de data-flow (également problème de complexité)
- Prouver que le nombre de ϕ est suffisant pour être en forme SSA est non évident.

Difficultés liées à la fois au calcul des fonctions *et* à leur preuve.

Où insérer des ϕ ?

Question : Combien de ϕ faut-il insérer (et où ?)

Réponse traditionnelle : en chaque point du flot de contrôle où deux assignation différentes se rencontrent (pour la 1ère fois) (sur la frontière de domination) :
Forme SSA *minimale*

Problèmes

- Obtenir la frontière de domination n'est pas trivial : problème de data-flow (également problème de complexité)
- Prouver que le nombre de ϕ est suffisant pour être en forme SSA est non évident.

Difficultés liées à la fois au calcul des fonctions *et* à leur preuve.

Où insérer des ϕ ?

Question : Combien de ϕ faut-il insérer (et où ?)

Réponse traditionnelle : en chaque point du flot de contrôle où deux assignation différentes se rencontrent (pour la 1ère fois) (sur la frontière de domination) :
Forme SSA *minimale*

Problèmes

- Obtenir la frontière de domination n'est pas trivial : problème de data-flow (également problème de complexité)
- Prouver que le nombre de ϕ est suffisant pour être en forme SSA est non évident.

Difficultés liées à la fois au calcul des fonctions *et* à leur preuve.

Autre approche

Algorithme (très) simple

On insère une instruction ϕ pour chaque variable déjà définie au début d'un bloc : forme SSA *"semi-maximale"*.
Ensuite on supprime les instructions inutiles.

Autre approche

Algorithme (très) simple

On insère une instruction ϕ pour chaque variable déjà définie au début d'un bloc : forme SSA *"semi-maximale"*.
Ensuite on supprime les instructions inutiles.

Résultat théorique

Si le graphe de flot de contrôle est réductible (ce qui est notre cas), une fois la phase de suppression effectuée on est en forme SSA minimale.

Autre approche

Algorithme (très) simple

On insère une instruction ϕ pour chaque variable déjà définie au début d'un bloc : forme SSA *"semi-maximale"*.
Ensuite on supprime les instructions inutiles.

Résultat pratique

Pas beaucoup plus lent que les algorithmes mettant en jeu la frontière de domination. Et des améliorations sont possibles.

Autre approche

Algorithme (très) simple

On insère une instruction ϕ pour chaque variable déjà définie au début d'un bloc : forme SSA *"semi-maximale"*.
Ensuite on supprime les instructions inutiles.

Inconvénient

La frontière de domination n'est pas calculée, et ne peut pas servir pour les optimisations.

Syntaxe du langage

IMP avec blocs

Variation sur le langage IMP, avec deux types d'états (registres et mémoires), des blocs, et les fonctions ϕ.

Exemples d'éléments syntaxiques

Calculs entre registres : $r \leftarrow 2 \star r$
Appels de fonctions : $r \leftarrow F(3, r_1, r_2)$
Accès mémoire : $[r] \leftarrow 5$, $r \leftarrow [r']$
Saut conditionnel : $\rightsquigarrow r^? : l_T, l_F$
Renvoi de résultat : $\rightarrow\!| \; r$

Sémantique opérationnelle

$$\frac{\mathcal{R}(\mu_1) = v_1 \qquad \mathcal{R}(\mu_2) = v_2 \qquad v_1 \star v_2 = v}{\langle \mathcal{R}, \mathcal{M}, l\_, r \leftarrow \mu_1 \star \mu_2 \rangle \rightarrow \langle \mathcal{R}[r := v], \mathcal{M} \rangle} \text{ SOp}$$

$$\frac{\forall n,\ \mathcal{R}(\mu_n) = v_n \qquad \langle \mathcal{M}, (v_1, ..., v_n), F \rangle \Mapsto \langle \mathcal{M}', v \rangle}{\langle \mathcal{R}, \mathcal{M}, l\_, r \leftarrow F(\mu_1, ..., \mu_n) \rangle \rightarrow \langle \mathcal{R}[r := v], \mathcal{M}' \rangle} \text{ SCall}$$

$$\frac{\mathcal{R}(\mu) = v \neq 0}{\langle \mathcal{R}, \rightsquigarrow \mu^? : l_T, l_F \rangle \rightsquigarrow l_T} \text{ SCondJumpT}$$

Instructions ϕ

$$i_\phi = \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \leftarrow \phi \begin{pmatrix} l_1 & \dots & l_m \\ \mu_{1,1} & \dots & \mu_{1,m} \\ \vdots & \ddots & \vdots \\ \mu_{n,1} & \dots & \mu_{n,m} \end{pmatrix}$$

$$\frac{\forall p,\ \mathcal{R}(\mu_{p,q}) = v_p}{\langle \mathcal{R}, \mathcal{M}, l_q, i_\phi \rangle \rightarrow \langle \mathcal{R}[r_1 := v_1, ..., r_n := v_n], \mathcal{M} \rangle}\ \textsc{SPhi}$$

Remarque

Le parallélisme des instructions ϕ est rendu explicite.

Instructions ϕ

$$i_\phi = \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \leftarrow \phi \begin{pmatrix} l_1 & \dots & l_m \\ \mu_{1,1} & \dots & \mu_{1,m} \\ \vdots & \ddots & \vdots \\ \mu_{n,1} & \dots & \mu_{n,m} \end{pmatrix}$$

$$\frac{\forall p,\ \mathcal{R}(\mu_{p,q}) = v_p}{\langle \mathcal{R}, \mathcal{M}, l_q, i_\phi \rangle \rightarrow \langle \mathcal{R}[r_1 := v_1, ..., r_n := v_n], \mathcal{M} \rangle}\ \textsc{SPhi}$$

Remarque

Le parallélisme des instructions ϕ est rendu explicite.

Sortie de la forme SSA

Il faut remplacer les fonctions ϕ par une suite d'instructions équivalentes.

Multi-copie

On suppose disposer d'une fonction $\textsc{MultiCopie}(r_1, ..., r_n, v_1, ..., v_n)$ qui effectue la copie *en parallèle* de v_1 dans r_1, ..., v_n dans r_n.

Insertion des copies

On ajoute un nouveau bloc sur chaque arête arrivant sur une fonction ϕ, et on insère dans ce bloc une instruction de multi-copie.

Sortie de la forme SSA

Il faut remplacer les fonctions ϕ par une suite d'instructions équivalentes.

Multi-copie

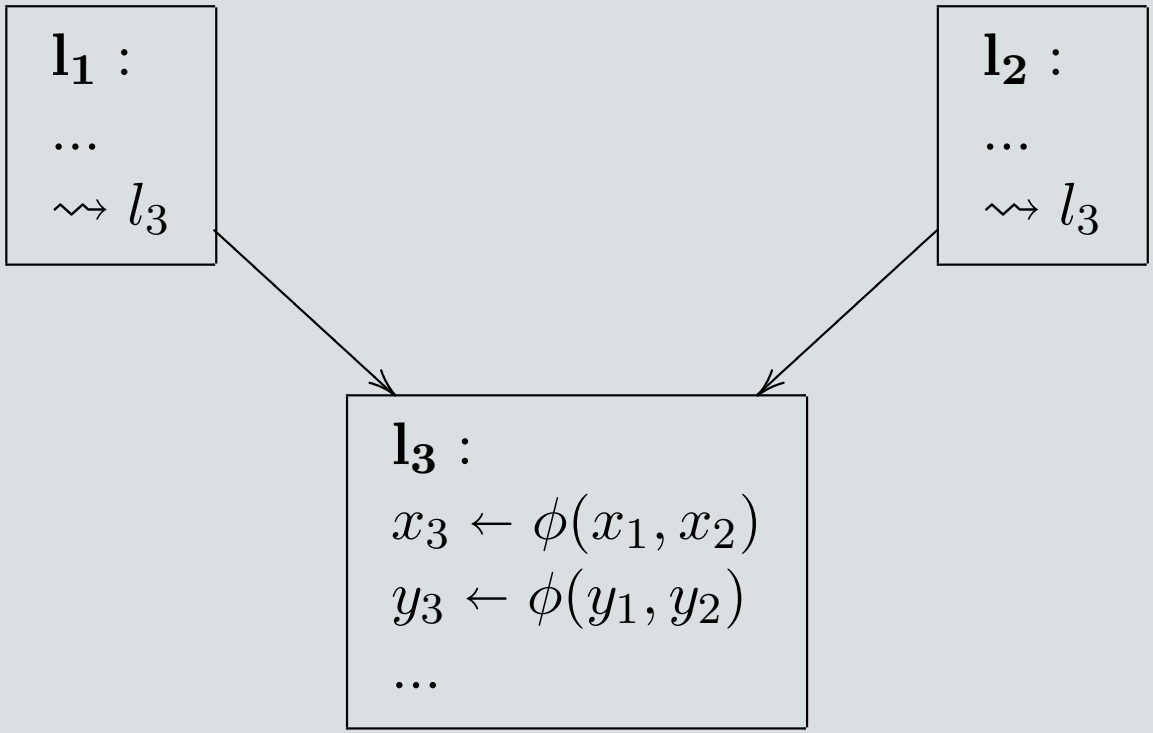
On suppose disposer d'une fonction $\textsc{MultiCopie}(r_1, ..., r_n, v_1, ..., v_n)$ qui effectue la copie *en parallèle* de v_1 dans r_1, ..., v_n dans r_n.

Insertion des copies

On ajoute un nouveau bloc sur chaque arête arrivant sur une fonction ϕ, et on insère dans ce bloc une instruction de multi-copie.

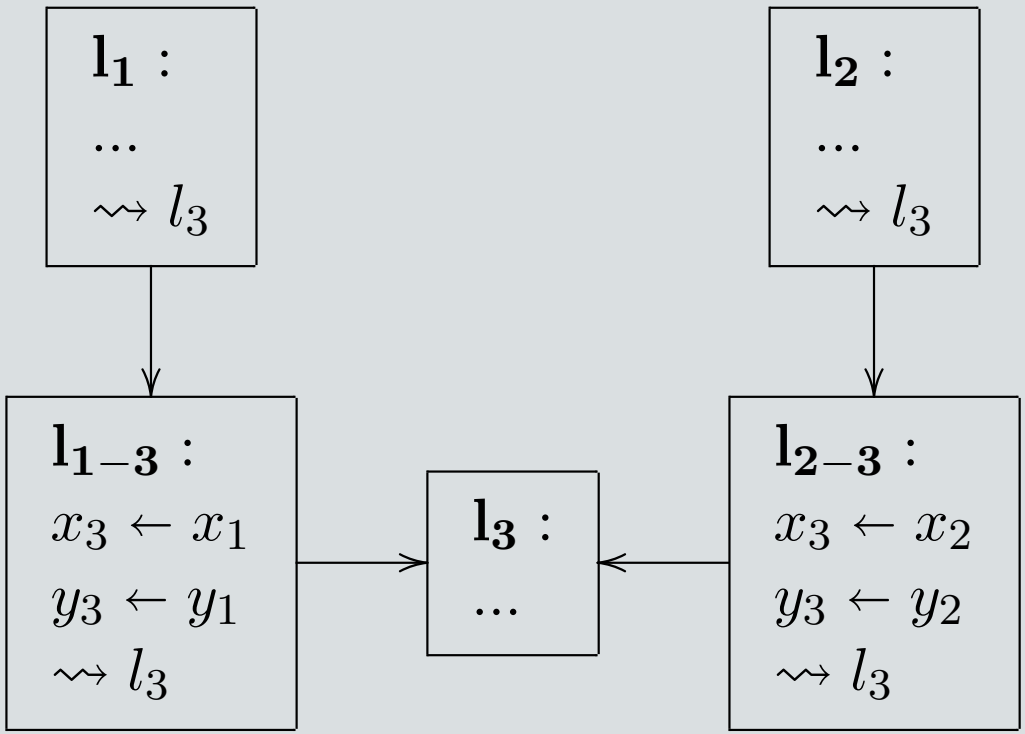
Exemple de sortie de la forme SSA

Example



Exemple de sortie de la forme SSA

Example



Plan

Optimisations envisageables

Les optimisations actuellement présentes dans Concert sont toutes faites par des analyses de Work-Flow différentes.

Optimisations en forme SSA

- Propagation de constantes.
  En forme SSA, effectuée avec une liste de tâches.
- Élimination des sous-expressions communes.
- Live range splitting.
- Suppression de code mort.
- Propagation de copies (non effectuée par le compilateur actuel).

Optimisations envisageables

Les optimisations actuellement présentes dans Concert sont toutes faites par des analyses de Work-Flow différentes.

Optimisations en forme SSA

- Propagation de constantes.
- Élimination des sous-expressions communes.
  Nécessite actuellement une structure de données complexe, pour maintenir des classes d'équivalence.
  En SSA, peut être fait en une passe sur chaque bloc de base ; également plus performant "en général".
- Live range splitting.
- Suppression de code mort.
- Propagation de copies (non effectuée par le compilateur actuel).

Optimisations envisageables

Les optimisations actuellement présentes dans Concert sont toutes faites par des analyses de Work-Flow différentes.

Optimisations en forme SSA

- Propagation de constantes.
- Élimination des sous-expressions communes.
- Live range splitting.
  Par construction de SSA
- Suppression de code mort.
- Propagation de copies (non effectuée par le compilateur actuel).

Optimisations envisageables

Les optimisations actuellement présentes dans Concert sont toutes faites par des analyses de Work-Flow différentes.

Optimisations en forme SSA

- Propagation de constantes.
- Élimination des sous-expressions communes.
- Live range splitting.
- Suppression de code mort.
  En parallèle de la propagation de constantes
- Propagation de copies (non effectuée par le compilateur actuel).

Optimisations envisageables

Les optimisations actuellement présentes dans Concert sont toutes faites par des analyses de Work-Flow différentes.

Optimisations en forme SSA

- Propagation de constantes.
- Élimination des sous-expressions communes.
- Live range splitting.
- Suppression de code mort.
- Propagation de copies (non effectuée par le compilateur actuel).
  En parallèle de la propagation de constantes

Optimisations envisageables

Les optimisations actuellement présentes dans Concert sont toutes faites par des analyses de Work-Flow différentes.

Optimisations en forme SSA

- Propagation de constantes.
- Élimination des sous-expressions communes.
- Live range splitting.
- Suppression de code mort.
- Propagation de copies (non effectuée par le compilateur actuel).

Algorithme par liste de tâches

Algorithme générique

$\textsc{WorkList}(F)$

1 $F' \leftarrow F$
2 $W \leftarrow \tau(F)$
3 **while** $W \neq \emptyset$
4 **do** $\{\tau\} \cup W' \leftarrow W$
5 **if** $P(F', \tau)$
6 **then** $F' \leftarrow T(F, \tau)$
7 $W \leftarrow W' \cup C(F, \tau)$
8 **else** $W \leftarrow W'$
9 **return** F'

Suppression de ϕ

Instructions ϕ inutiles

Certaines instructions ϕ sont inutiles, et on peut les supprimer pour arriver en forme SSA minimale.

On considère une instruction ϕ contenant une ligne de la forme $x \leftarrow \phi(v_1, ..., v_n)$ avec $\{v_1, ..., v_n\} = \{x, y\}$.
On peut la remplacer par la même instruction ϕ, sans la ligne mettant en jeu x, suivie d'une affectation $x \leftarrow y$.

Résultat fondamental, qui a été prouvé.

Suppression de ϕ

Instructions ϕ inutiles

Certaines instructions ϕ sont inutiles, et on peut les supprimer pour arriver en forme SSA minimale.

On considère une instruction ϕ contenant une ligne de la forme $x \leftarrow \phi(v_1, ..., v_n)$ avec $\{v_1, ..., v_n\} = \{x, y\}$.
On peut la remplacer par la même instruction ϕ, sans la ligne mettant en jeu x, suivie d'une affectation $x \leftarrow y$.

Les affectations $x \leftarrow x$ sont redondantes. Dans toutes les autres branches, on a $x \leftarrow y$, et donc x est un alias pour y.

Résultat fondamental, qui a été prouvé.

Suppression de ϕ

Instructions ϕ inutiles

Certaines instructions ϕ sont inutiles, et on peut les supprimer pour arriver en forme SSA minimale.

On considère une instruction ϕ contenant une ligne de la forme $x \leftarrow \phi(v_1, ..., v_n)$ avec $\{v_1, ..., v_n\} = \{x, y\}$.
On peut la remplacer par la même instruction ϕ, sans la ligne mettant en jeu x, suivie d'une affectation $x \leftarrow y$.

Résultat fondamental, qui a été prouvé.

Utilisation des propriétés de la forme SSA

Unicité de l'assignation

Chaque variable n'est assignée qu'une seule fois. La propagation de constantes ou de copies deviennent presque triviales.

Lemme de substitution

En cas d'affectation $x = v$, on peut remplacer x par v dans tout la fonction.

Élimination d'assignation morte

S'il existe une assignation $x = v$ ou $x = v_1 \star v_2$, et que x n'est pas utilisé, on peut supprimer l'assignation.

Utilisation des propriétés de la forme SSA

Unicité de l'assignation

Chaque variable n'est assignée qu'une seule fois. La propagation de constantes ou de copies deviennent presque triviales.

Lemme de substitution

En cas d'affectation $x = v$, on peut remplacer x par v dans tout la fonction.

Élimination d'assignation morte

S'il existe une assignation $x = v$ ou $x = v_1 \star v_2$, et que x n'est pas utilisé, on peut supprimer l'assignation.

Conclusion

Conclusion

Objectifs du stage atteints : on dispose d'une sémantique formelle de SSA, d'optimisations prouvées, et d'algorithmes permettant de passer d'une forme à l'autre.

Passage vers la preuve formelle restant à faire, même si les preuves "papier" ont été faites dans cette optique.

Questions

?

Conclusion

Conclusion

Objectifs du stage atteints : on dispose d'une sémantique formelle de SSA, d'optimisations prouvées, et d'algorithmes permettant de passer d'une forme à l'autre.

Passage vers la preuve formelle restant à faire, même si les preuves "papier" ont été faites dans cette optique.

Questions

?