Traduction bidirectionnelle entre un dialecte du C et un λ-calcul impératif

Guillaume Bertholon Arthur Charguéraud

Université de Strasbourg ICUBE CNRS Inria

JFLA 2025

Multiplication de matrices spécialisée pour 1024×1024

```
void mm1024(float* C, float* A, float* B) {
  for (int i = 0; i < 1024; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        sum += A[1024 * i + k] * B[1024 * k + j];
      }
      C[1024 * i + j] = sum;
    }
  }
}
```

Multiplication de matrices spécialisée pour 1024×1024

```
void mm1024(float* C, float* A, float* B) {
  for (int i = 0; i < 1024; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        sum += A[1024 * i + k] * B[1024 * k + j];
      }
      C[1024 * i + j] = sum;
    }
  }
}
```

150× plus rapide
sur Intel i7-8665U, 4 cores, AVX2

Multiplication de matrice optimisée

```c
void mm1024(float* C, float* A, float* B) {
  float* const pB =
    malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++)
    for (int bk = 0; bk < 256; bk++)
      for (int k = 0; k < 4; k++)
        for (int j = 0; j < 32; j++)
          pB[32768 * bj + 128 * bk + 32 * k + j] =
              B[1024 * (4 * bk + k) + 32 * bj + j];

  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* const sum =
        malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          sum[32 * i + j] = 0.f;
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++)
            s[j] += A[1024 * (32*bi + i) + 4*bk + 0]
                  * pB[32768*bj + 128*bk + 32*0 + j];
          // [même boucle avec k = 1, 2, 3]
          memcpy(&sum[32 * i], s, sizeof(float[32]));
        }
      }
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          C[1024 * (32 * bi + i) + 32 * bj + j]
            = sum[32 * i + j];
      free(sum);
    }
  }
  free(pB); }
```

tuilage, transposition, parallélisation, vectorisation, memcopy hoisting

Multiplication de matrice optimisée

```c
void mm1024(float* C, float* A, float* B) {
  float* const pB =
    malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++)
    for (int bk = 0; bk < 256; bk++)
      for (int k = 0; k < 4; k++)
        for (int j = 0; j < 32; j++)
          pB[32768 * bj + 128 * bk + 32 * k + j] =
              B[1024 * (4 * bk + k) + 32 * bj + j];

  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* const sum =
        malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          sum[32 * i + j] = 0.f;
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++)
            s[j] += A[1024 * (32*bi + i) + 4*bk + 0]
                  * pB[32768*bj + 128*bk + 32*0 + j];
          // [même boucle avec k = 1, 2, 3]
          memcpy(&sum[32 * i], s, sizeof(float[32]));
        }
      }
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          C[1024 * (32 * bi + i) + 32 * bj + j]
            = sum[32 * i + j];
      free(sum);
    }
  }
free(pB); }
```

tuilage, transposition, parallélisation, vectorisation, memcpy hoisting

Multiplication de matrice optimisée

```c
void mm1024(float* C, float* A, float* B) {
  float* const pB =
    malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++)
    for (int bk = 0; bk < 256; bk++)
      for (int k = 0; k < 4; k++)
        for (int j = 0; j < 32; j++)
          pB[32768 * bj + 128 * bk + 32 * k + j] =
              B[1024 * (4 * bk + k) + 32 * bj + j];

  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* const sum =
        malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          sum[32 * i + j] = 0.f;
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++)
            s[j] += A[1024 * (32*bi + i) + 4*bk + 0]
                  * pB[32768*bj + 128*bk + 32*0 + j];
          // [même boucle avec k = 1, 2, 3]
          memcpy(&sum[32 * i], s, sizeof(float[32]));
        }
      }
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          C[1024 * (32 * bi + i) + 32 * bj + j]
            = sum[32 * i + j];
      free(sum);
    }
  }
  free(pB); }
```

tuilage, transposition, parallélisation, vectorisation, memcpy hoisting

Multiplication de matrice optimisée

```c
void mm1024(float* C, float* A, float* B) {
  float* const pB =
    malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++)
    for (int bk = 0; bk < 256; bk++)
      for (int k = 0; k < 4; k++)
        for (int j = 0; j < 32; j++)
          pB[32768 * bj + 128 * bk + 32 * k + j] =
              B[1024 * (4 * bk + k) + 32 * bj + j];

  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* const sum =
        malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          sum[32 * i + j] = 0.f;
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++)
            s[j] += A[1024 * (32*bi + i) + 4*bk + 0]
                  * pB[32768*bj + 128*bk + 32*0 + j];
          // [même boucle avec k = 1, 2, 3]
          memcpy(&sum[32 * i], s, sizeof(float[32]));
        }
      }
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          C[1024 * (32 * bi + i) + 32 * bj + j]
            = sum[32 * i + j];
      free(sum);
    }
  }
  free(pB); }
```

tuilage, transposition, parallélisation, vectorisation, memcpy hoisting

Multiplication de matrice optimisée

```c
void mm1024(float* C, float* A, float* B) {
  float* const pB =
    malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++)
    for (int bk = 0; bk < 256; bk++)
      for (int k = 0; k < 4; k++)
        for (int j = 0; j < 32; j++)
          pB[32768 * bj + 128 * bk + 32 * k + j] =
            B[1024 * (4 * bk + k) + 32 * bj + j];

  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* const sum =
        malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          sum[32 * i + j] = 0.f;
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++)
            s[j] += A[1024 * (32*bi + i) + 4*bk + 0]
                  * pB[32768*bj + 128*bk + 32*0 + j];
          // [même boucle avec k = 1, 2, 3]
          memcpy(&sum[32 * i], s, sizeof(float[32]));
        }
      }
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          C[1024 * (32 * bi + i) + 32 * bj + j]
            = sum[32 * i + j];
      free(sum);
    }
  }
  free(pB); }
```

tuilage, transposition, parallélisation, vectorisation, memcopy hoisting

Multiplication de matrice optimisée

```c
void mm1024(float* C, float* A, float* B) {
  float* const pB =
    malloc(sizeof(float[32][256][4][32]));
  #pragma omp parallel for
  for (int bj = 0; bj < 32; bj++)
    for (int bk = 0; bk < 256; bk++)
      for (int k = 0; k < 4; k++)
        for (int j = 0; j < 32; j++)
          pB[32768 * bj + 128 * bk + 32 * k + j] =
              B[1024 * (4 * bk + k) + 32 * bj + j];

  #pragma omp parallel for
  for (int bi = 0; bi < 32; bi++) {
    for (int bj = 0; bj < 32; bj++) {
      float* const sum =
        malloc(sizeof(float[32][32]));
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          sum[32 * i + j] = 0.f;
      for (int bk = 0; bk < 256; bk++) {
        for (int i = 0; i < 32; i++) {
          float s[32];
          memcpy(s, &sum[32 * i], sizeof(float[32]));
          #pragma omp simd
          for (int j = 0; j < 32; j++)
            s[j] += A[1024 * (32*bi + i) + 4*bk + 0]
                  * pB[32768*bj + 128*bk + 32*0 + j];
          // [même boucle avec k = 1, 2, 3]
          memcpy(&sum[32 * i], s, sizeof(float[32]));
        }
      }
      for (int i = 0; i < 32; i++)
        for (int j = 0; j < 32; j++)
          C[1024 * (32 * bi + i) + 32 * bj + j]
            = sum[32 * i + j];
      free(sum);
    }
  }
  free(pB); }
```

tuilage, transposition, parallélisation, vectorisation, memcopy hoisting

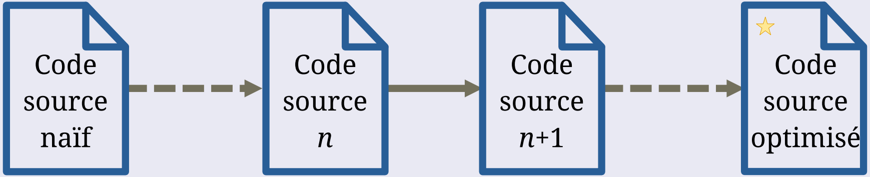
Comment obtenir un code optimisé

Deux options pour obtenir du code haute performance

- Écrire le code optimisé à la main ⇒ Long et risque d'erreur
- Utiliser un compilateur spécialisé comme Halide ou TVM ⇒ Perte de généralité

Une nouvelle option : OptiTrust

- Transformations source-à-source interactive pilotée par l'utilisateur
- Séquence de transformation décrite dans un script OCaml



Exemple de script OptiTrust

Script de transformation

```
Function.inline_def [cFunDef "mm"];
Loop.tile (int 32) ~index:"bi" ~bound:TileDivides
  [cFor "i"];
let tile (loop_id, tile_size) =
  Loop.tile (int tile_size) ~index:("b" ^ loop_id)
    ~bound:TileDivides [cFor loop_id] in
List.iter tile [("j", 32); ("k", 4)];
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
      [cPlusEq [cVar "sum"]];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
    ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s"
    ~copy_dims:1 [cFor ~body:[cPlusEq [cVar "sum"]] "k"
         ];
Matrix.elim_mops [];
Loop.unroll [cFor ~body:[cPlusEq [cVar "s"]] "k"];
Omp.simd [nbMulti; cFor ~body:[cPlusEq [cVar "s"]] "j"];
Omp.parallel_for [nbMulti; cFunBody "mm1024"; cStrict;
     cFor ""];
```

Retour intéractif

```
void mm(float* C, float* A, float* B,
        int m, int n, int p) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      float sum = 0.f;
      for (int k = 0; k < p; k++) {
        sum += A[p * i + k]
               * B[n * k + j];
      }
      C[n * i + j] = sum;
    }
  }
}

void mm1024(float* C, float* A, float* B) {
  mm(C, A, B, 1024, 1024, 1024);
}
```

Exemple de script OptiTrust

Script de transformation

```
Function.inline_def [cFunDef "mm"];
Loop.tile (int 32) ~index:"bi" ~bound:TileDivides
  [cFor "i"];
let tile (loop_id, tile_size) =
  Loop.tile (int tile_size) ~index:("b" ^ loop_id)
    ~bound:TileDivides [cFor loop_id] in
List.iter tile [("j", 32); ("k", 4)];
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
      [cPlusEq [cVar "sum"]];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
    ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s"
    ~copy_dims:1 [cFor ~body:[cPlusEq [cVar "sum"]] "k"
         ];
Matrix.elim_mops [];
Loop.unroll [cFor ~body:[cPlusEq [cVar "s"]] "k"];
Omp.simd [nbMulti; cFor ~body:[cPlusEq [cVar "s"]] "j"];
Omp.parallel_for [nbMulti; cFunBody "mm1024"; cStrict;
     cFor ""];
```

Retour intéractif

```
void mm(float* C, float* A, float* B,
         int m, int n, int p) {
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      float sum = 0.f;
      for (int k = 0; k < p; k++) {
        sum += A[p * i + k]
               * B[n * k + j];
      }
      C[n * i + j] = sum;
    }
  }
}

void mm1024(float* C, float* A, float* B) {
  mm(C, A, B, 1024, 1024, 1024);
}
```

Exemple de script OptiTrust

Script de transformation

```
Function.inline_def [cFunDef "mm"];
Loop.tile (int 32) ~index:"bi" ~bound:TileDivides
  [cFor "i"];
let tile (loop_id, tile_size) =
  Loop.tile (int tile_size) ~index:("b" ^ loop_id)
    ~bound:TileDivides [cFor loop_id] in
List.iter tile [("j", 32); ("k", 4)];
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
      [cPlusEq [cVar "sum"]];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
    ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s"
    ~copy_dims:1 [cFor ~body:[cPlusEq [cVar "sum"]] "k"
        ];
Matrix.elim_mops [];
Loop.unroll [cFor ~body:[cPlusEq [cVar "s"]] "k"];
Omp.simd [nbMulti; cFor ~body:[cPlusEq [cVar "s"]] "j"];
Omp.parallel_for [nbMulti; cFunBody "mm1024"; cStrict;
     cFor ""];
```

Retour intéractif

```
void mm1024(float* C, float* A, float* B) {
  for (int i = 0; i < 1024; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        sum += A[1024 * i + k]
                * B[1024 * k + j];
      }
      C[1024 * i + j] = sum;
    }
  }
}
```

Exemple de script OptiTrust

Script de transformation

```
Function.inline_def [cFunDef "mm"];
Loop.tile (int 32) ~index:"bi" ~bound:TileDivides
  [cFor "i"];
let tile (loop_id, tile_size) =
  Loop.tile (int tile_size) ~index:("b" ^ loop_id)
   ~bound:TileDivides [cFor loop_id] in
List.iter tile [("j", 32); ("k", 4)];
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
      [cPlusEq [cVar "sum"]];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
   ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s"
   ~copy_dims:1 [cFor ~body:[cPlusEq [cVar "sum"]] "k"
        ];
Matrix.elim_mops [];
Loop.unroll [cFor ~body:[cPlusEq [cVar "s"]] "k"];
Omp.simd [nbMulti; cFor ~body:[cPlusEq [cVar "s"]] "j"];
Omp.parallel_for [nbMulti; cFunBody "mm1024"; cStrict;
     cFor ""];
```

Retour intéractif

```
void mm1024(float* C, float* A, float* B) {
  for (int i = 0; i < 1024; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        sum += A[1024 * i + k]
               * B[1024 * k + j];
      }
      C[1024 * i + j] = sum;
    }
  }
}
```

Exemple de script OptiTrust

Script de transformation

```
Function.inline_def [cFunDef "mm"];
Loop.tile (int 32) ~index:"bi" ~bound:TileDivides
  [cFor "i"];
let tile (loop_id, tile_size) =
  Loop.tile (int tile_size) ~index:("b" ^ loop_id)
    ~bound:TileDivides [cFor loop_id] in
List.iter tile [("j", 32); ("k", 4)];
Loop.reorder_at ~order:["bi"; "bj"; "bk"; "i"; "k"; "j"]
      [cPlusEq [cVar "sum"]];
Loop.hoist_expr ~dest:[tBefore; cFor "bi"] "pB"
    ~indep:["bi"; "i"] [cArrayRead "B"];
Matrix.stack_copy ~var:"sum" ~copy_var:"s"
    ~copy_dims:1 [cFor ~body:[cPlusEq [cVar "sum"]] "k"
         ];
Matrix.elim_mops [];
Loop.unroll [cFor ~body:[cPlusEq [cVar "s"]] "k"];
Omp.simd [nbMulti; cFor ~body:[cPlusEq [cVar "s"]] "j"];
Omp.parallel_for [nbMulti; cFunBody "mm1024"; cStrict;
     cFor ""];
```

Retour intéractif

```
void mm1024(float* C, float* A, float* B) {
  for (int bi = 0; bi < 32; bi++) {
  for (int i = 0; i < 32; i++) {
    for (int j = 0; j < 1024; j++) {
      float sum = 0.f;
      for (int k = 0; k < 1024; k++) {
        sum += A[1024 * (32 * bi + i) + k]
               * B[1024 * k + j];
      }}
      C[1024 * (32 * bi + i) + j] = sum;
    }}
  }}
}
```

Fonctionnement des transformations

En général, les transformations suivent le schéma suivant :

- Mettre à jour des analyses statiques sur le code (ex : typage)
- Trouver le sous-terme où appliquer la transformation
- Analyser l'AST au point de transformation
- Produire un nouvel AST
- Afficher ce nouvel AST à l'utilisateur

Du point de vue de l'utilisateur les transformations manipulent du code C.

Pourquoi transformer du C mais pas directement ?

Avantage d'un langage comme le C

Syntaxe et sémantique bien connue par les programmeurs HPC

Difficultés pour l'implémentation des transformations

- Variables mutables par défaut
  - Complique les substitutions et impose de vérifier s'il y a eu une modification à chaque fois
  - Difficultés pour exprimer des assertions
- Règles d'évaluation différentes pour les left-values
  - `x = x + 1`
  - `f(&k, k)`

Deux langages et une traduction bidirectionnelle

Solution pour éviter les problèmes avec les left-values : utiliser du λ-calcul.

OptiC

- Dialecte du C (sous-ensemble + ajouts mineurs)
- Variables mutables et left-values
- Utilisé pour les interactions avec l'utilisateur

Optiλ

- λ-calcul impératif
- Toute mutation est derrière un pointeur
- Manipulé par les transformations

Traduction bidirectionnelle qui préserve le code en cas d'aller-retour

L'approche s'applique à tout outil interactif qui a besoin d'un langage interne simple.
Permet de changer le langage d'interaction facilement.

Idée principale de la traduction

Diviser les variables OptiC en deux catégories :

- Variables pures
  - Variables déclarées `const`
  - Arguments de fonction
  - Nom des fonctions elles même

  Interdiction de prendre leur adresse ou de les modifier
- Variables non pures
  - Toutes les autres

  Interdiction de les utiliser dans les assertions ou le typage

Exemple de traduction

Uniquement des variables pures

```
int norm2(int x, int y){
  const int xsq = x * x;
  const int ysq = y * y;
  const int res = xsq + ysq;
  return res;
}
```

$$
\begin{array}{l}
\mathbf{let}\ \mathtt{norm2} = \mathbf{fun}(\mathtt{x} : \mathsf{int}, \mathtt{y} : \mathsf{int}) \mapsto \{ \\
\quad \mathbf{let}_{\mathsf{int}}\ \mathtt{xsq} = \mathsf{mul}(\mathtt{x}, \mathtt{x}); \\
\quad \mathbf{let}_{\mathsf{int}}\ \mathtt{ysq} = \mathsf{mul}(\mathtt{y}, \mathtt{y}); \\
\quad \mathbf{let}_{\mathsf{int}}\ \mathtt{res} = \mathsf{add}(\mathtt{xsq}, \mathtt{ysq}); \\
\quad \mathtt{res} \\
\};
\end{array}
$$

Exemple de traduction

Avec une variable non pure

```
int norm2Acc(int x, int y){
  int acc;
  acc = x * x;
  acc += y * y;
  return acc;

}
```

$$
\begin{array}{l}
\textbf{let}\ \texttt{norm2Acc} = \textbf{fun}(\texttt{x} : \mathsf{int}, \texttt{y} : \mathsf{int}) \mapsto \{ \\
\quad \textbf{let}_{\mathsf{ptr(int)}}\ \texttt{acc} = \mathsf{stackCell}_{\mathsf{int}}(); \\
\quad \mathsf{set}(\texttt{acc}, \mathsf{mul}(\texttt{x}, \texttt{x})); \\
\quad \mathsf{inplaceAdd}(\texttt{acc}, \mathsf{mul}(\texttt{y}, \texttt{y})); \\
\quad \textbf{let}_{\mathsf{int}}\ \texttt{res} = \mathsf{get}(\texttt{acc}); \\
\quad \texttt{res} \\
\};
\end{array}
$$

Plan du reste de l'exposé

1. Particularités des langages Optiλ et OptiC
2. Détails de la traduction OptiC $\mapsto$ Optiλ
3. Traduction inverse et annotations de style
4. Propriétés formelles

Particularités d'Optiλ

Optiλ est un λ-calcul avec :

- Cellules mutables
  - Opérateurs get(t), set(t_1, t_2), heapCell() et free()
- Séquences à plat
  - Sans valeur de retour : $\{t_1; \dots; t_n;\}$
  - Avec valeur de retour : $\{t_1; \dots; \mathbf{let}\ x = t_x; \dots; t_n; x\}$
  - $\mathbf{let}\ x = t$ est une instruction
  - Aide pour le ciblage d'instructions contiguës
  - Permet d'exprimer des assertions sur la valeur de retour dans la séquence elle-même
- Cellules détruites à la fin de leur séquence
  - $\{t_1; \dots; \mathbf{let}\ x = \text{stackCell}(); \dots; t_n; r\}$
- Sémantique call-by-value
  - Notamment, il n'y a pas d'opérateur `&&` ou `||`
  - $f(t_1, \dots, t_n)$ évalue les t_i de manière concurrente
- Annotations sur les noeuds d'AST
  - Exemple : résultats des analyses statiques

Particularités d'OptiC

OptiC est un sous-ensemble du C avec quelques extensions :

- Syntaxe pour supporter toutes les constructions d'Optiλ
  - Statement-exprs : `int i = ({ int x = f(y); x + 1; });`
  - Clotures : `fun<int(int)> f = [&](int a){ return a + 1; };`
- Ordre plus strict d'évaluation des opérateurs
  - Exemple : `i = i++;` bien défini en OptiC (UB en C)
- Pas d'adresse sur les variables pures

Quelques limites supplémentaires dans OptiTrust :

- Idéalisation des entiers (taille infinie) et des flottants (réels)
- Pas encore de support de `break`, `continue` et `return` en position non-terminale

2 modes de traduction pour traiter les prises d'adresse : $\lfloor u \rfloor$ et $\lfloor u \rfloor^{\&}$.

`int y;`	$\longleftrightarrow$	$\textbf{let}_{\mathsf{ptr(int)}}\ \mathtt{y} = \mathsf{stackCell}_{\mathsf{int}}();$
`y = 6;`	$\longleftrightarrow$	$\mathsf{set}(\mathtt{y}, 6);$
`f(&y);`	$\longleftrightarrow$	$\mathtt{f}(\mathtt{y});$
`int* const z = malloc(sizeof(int));`	$\longleftrightarrow$	$\textbf{let}_{\mathsf{ptr(int)}}\ \mathtt{z} = \mathsf{heapCell}_{\mathsf{int}}();$
`*z = *z + 2;`	$\longleftrightarrow$	$\mathsf{set}(\mathtt{z}, \mathsf{get}(\mathtt{z}) + 2);$
`int* const p = &y;`	$\longleftrightarrow$	$\textbf{let}_{\mathsf{ptr(int)}}\ \mathtt{p} = \mathtt{y};$
`*p = *p + 2`	$\longleftrightarrow$	$\mathsf{set}(\mathtt{p}, \mathsf{get}(\mathtt{p}) + 2);$
`int* q = &y;`	$\longleftrightarrow$	$\textbf{let}_{\mathsf{ptr(ptr(int))}}\ \mathtt{q} = \mathsf{ref}_{\mathsf{ptr(int)}}(\mathtt{y});$
`q = z;`	$\longleftrightarrow$	$\mathsf{set}(\mathtt{q}, \mathtt{z});$
`*q = *q + 2;`	$\longleftrightarrow$	$\mathsf{set}(\mathsf{get}(\mathtt{q}), \mathsf{get}(\mathsf{get}(\mathtt{q})) + 2);$

Parcours top-down, on accumule le contexte Π des variables pures

$$
\begin{array}{lcl}
\lfloor u \rfloor^{\&} & = & t \quad \text{where } \lfloor u \rfloor \text{ is (guaranteed to be) of the form get}(t) \\
\lfloor x \rfloor & = & \left| \begin{array}{ll} x & \text{if } x \in \Pi \\ \text{get}(x) & \text{otherwise} \end{array} \right. \\
\lfloor \& u \rfloor & = & \lfloor u \rfloor^{\&} \\
\lfloor * u \rfloor & = & \text{get}(\lfloor u \rfloor) \\
\lfloor u_1 \texttt{ = } u_2 \rfloor & = & \text{set}(\lfloor u_1 \rfloor^{\&}, \lfloor u_2 \rfloor) \\
\lfloor u_0(u_1, \ldots, u_n) \rfloor & = & \lfloor u_0 \rfloor(\lfloor u_1 \rfloor, \ldots, \lfloor u_n \rfloor) \\
\lfloor T \textbf{ const } x = u \rfloor & = & \textbf{let}_{\lfloor T \rfloor}\ x = \lfloor u \rfloor \qquad (x \in \Pi) \\
\lfloor T\ x = u \rfloor & = & \textbf{let}_{\text{ptr}(\lfloor T \rfloor)}\ x = \text{ref}_{\lfloor T \rfloor}(\lfloor u \rfloor) \qquad (x \notin \Pi) \\
\lfloor T\ x \rfloor & = & \textbf{let}_{\text{ptr}(\lfloor T \rfloor)}\ x = \text{stackCell}_{\lfloor T \rfloor}() \qquad (x \notin \Pi) \\
\ldots & = & \ldots
\end{array}
$$

Traduction d'Optiλ vers OptiC

Extrait de la formalisation

$$
\begin{array}{lcl}
\lceil t \rceil^* & = & \left| \begin{array}{ll} u & \text{if } \lceil t \rceil \text{ is of the form } \&u \\ *\lceil t \rceil & \text{otherwise} \end{array} \right. \\[1em]
\lceil x \rceil & = & \left| \begin{array}{ll} x & \text{if } x \in \Pi \\ \&x & \text{otherwise} \end{array} \right. \\[1em]
\lceil \mathrm{get}(t) \rceil & = & \lceil t \rceil^* \\
\lceil \mathrm{set}(t_1, t_2) \rceil & = & \lceil t_1 \rceil^* = \lceil t_2 \rceil \\[0.5em]
\lceil t_0(t_1, \ldots, t_n) \rceil & = & \lceil t_0 \rceil(\lceil t_1 \rceil, \ldots, \lceil t_n \rceil) \\[0.5em]
\lceil \mathbf{let}_{\mathrm{ptr}(\tau)}\ x = \mathrm{stackCell}() \rceil & = & \lceil \tau \rceil\, x; \qquad (x \notin \Pi) \\
\lceil \mathbf{let}_{\mathrm{ptr}(\tau)}\ x = \mathrm{ref}(t) \rceil & = & \lceil \tau \rceil\, x = \lceil t \rceil; \qquad (x \notin \Pi) \\
\lceil \mathbf{let}_{\tau}\, x \ = \ t \rceil & = & \lceil \tau \rceil\, \texttt{const}\, x = \lceil t \rceil; \qquad (x \in \Pi) \\[1em]
\ldots & = & \ldots
\end{array}
$$

Préserver le code en cas d'aller-retour

Annotations de style

Certaines constructions OptiC ont le même encodage en Optiλ :

- on veut permettre à l'utilisateur de reconnaitre son code initial
- on utilise des annotations pour les distinguer

```c
if (p && *p == 0){
  *p = 1;
}
```

$$\textbf{if } (\textbf{if} \quad p \textbf{ then } \mathsf{eq}(\mathsf{get}(p), 0) \textbf{ else } \mathsf{false}) \textbf{ then } \{ \\ \quad \mathsf{set}(p, 1) \\ \} \textbf{ else } \{\}$$

```c
if (p ? (*p == 0): false){
  *p = 1;
} else {}
```

$$\textbf{if } (\textbf{if} \quad p \textbf{ then } \mathsf{eq}(\mathsf{get}(p), 0) \textbf{ else } \mathsf{false}) \textbf{ then } \{ \\ \quad \mathsf{set}(p, 1) \\ \} \textbf{ else } \{\}$$

Préserver le code en cas d'aller-retour

Annotations de style

Certaines constructions OptiC ont le même encodage en Optiλ :

- on veut permettre à l'utilisateur de reconnaitre son code initial
- on utilise des annotations pour les distinguer

```
if (p && *p == 0){
  *p = 1;
}
```

$$
\begin{array}{l}
\textbf{if } (\textbf{if}^{\&\&}\ p \textbf{ then } \mathsf{eq}(\mathsf{get}(p), 0) \textbf{ else } \mathsf{false}) \textbf{ then } \{ \\
\quad \mathsf{set}(p, 1) \\
\} \textbf{ else } \{\}^{\emptyset}
\end{array}
$$

```
if (p ? (*p == 0): false){
  *p = 1;
} else {}
```

$$
\begin{array}{l}
\textbf{if } (\textbf{if}^{?:}\ p \textbf{ then } \mathsf{eq}(\mathsf{get}(p), 0) \textbf{ else } \mathsf{false}) \textbf{ then } \{ \\
\quad \mathsf{set}(p, 1) \\
\} \textbf{ else } \{\}
\end{array}
$$

Propriétés formelles de la traduction

Une traduction aller-retour préserve l'AST

Motif superflu

`&*t` et `*&t` sont des motifs superflus.
On s'autorise à les remplacer par `t` lors d'un aller-retour.

Propriété d'aller-retour

u ne contient pas de motif superflu $\Rightarrow \lceil \lfloor u \rfloor \rceil = u$.

Limites

Tout n'est pas capturé dans l'AST :

- Commentaires
- Macros de préprocesseur
- Indentation et lignes vides

2 sémantiques à grand pas :

Sémantique OptiC

$u/^{s}_{m} \Downarrow o$ et $u/^{s}_{m} \Downarrow^{\&} o$

- environnement s : variables vers adresses
- mémoire m : adresses vers valeurs
- état de sortie o : err ou $v/^{s'}_{m'}$

Sémantique Optiλ

$t/^{\sigma}_{\mu} \Downarrow \omega$

- environnement σ : variables vers valeurs
- mémoire μ : adresses vers valeurs
- état de sortie ω : err ou $v/^{\sigma'}_{\mu'}$

Lier les deux sémantiques

- réadressage ρ : correspondance entre adresses dans OptiC et adresses dans Optiλ
- $(s, m) \sim_\rho (\sigma, \mu)$: l'état d'entrée Optiλ (s, m) correspond à l'état (σ, μ) en suivant ρ.
- $o \sim_\rho \omega$: l'état de sortie Optiλ o correspond à l'état ω en suivant ρ.

Propriétés formelles de la traduction

Les traductions préservent la sémantique

Correction de la traduction OptiC vers Optiλ

Pour tout :

- programme OptiC u,
- réadressage ρ,
- s, m, σ, μ tels que $(s, m) \sim_\rho (\sigma, \mu)$,
- o et ω tels que :
  - $(u/^s_m \Downarrow o) \wedge (\lfloor u \rfloor /^\sigma_\mu \Downarrow \omega)$, ou
  - $(u/^s_m \Downarrow^\& o) \wedge (\lfloor u \rfloor^\& /^\sigma_\mu \Downarrow \omega)$.

Il existe ρ' tel que $o \sim_{\rho'} \omega$.

Correction de la traduction Optiλ vers OptiC

Pour tout :

- programme Optiλ t,
- réadressage ρ,
- s, m, σ, μ tels que $(s, m) \sim_\rho (\sigma, \mu)$,
- ω tel que $t/^\sigma_\mu \Downarrow \omega$,
- o tel que :
  - $\lceil t \rceil /^s_m \Downarrow o$, ou
  - $\lceil t \rceil^* /^s_m \Downarrow^\& o$.

Il existe ρ' tel que $o \sim_{\rho'} \omega$.

Conclusion

Contribution

Une traduction bidirectionnelle entre OptiC et Optiλ

- Intéragir avec l'utilisateur dans une syntaxe familière
- Garder une sémantique simple en interne
- Préserver le code lors de l'aller-retour
- Permet de supporter d'autres langages facilement

Travaux futurs

- Preuve Coq/Rocq des propriétés formelles
- Lier la sémantique d'OptiC avec CompCert
- Ajouter le formalisme pour les boucles et les structures
- Traduire les tableaux multidimensionnels

Grammaire d'Optiλ

r	$:= \mid \emptyset \mid x$	resultat d'une séquence
t	$:= \mid x$	variables
	$\mid b \mid n$	valeurs booléennes et numériques
	$\mid \{t_1; \ldots; t_n; r\}$	séquence
	$\mid \textbf{let } x = t$	définition de variable
	$\mid \textbf{fun}(a_1, \ldots, a_n) \mapsto t$	λ-abstraction
	$\mid t_0(t_1, \ldots, t_n)$	appel de fonction
	$\mid \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2$	conditionelle
	$\mid \mathrm{add}(t_1, t_2) \mid \mathrm{inplaceAdd}(t_1, t_2) \mid \ldots$	opérations arithmétiques
	$\mid \mathrm{ignore}(t)$	primitive pour ignorer une valeur
	$\mid \mathrm{get}(t) \mid \mathrm{set}(t_1, t_2) \mid \mathrm{free}(t)$	primitives sur les cellules mémoire
	$\mid \mathrm{stackCell}() \mid \mathrm{ref}(t) \mid \mathrm{heapCell}()$	allocations de cellules mémoire

`int f(int n){ ... }`	$\longleftrightarrow$	$\mathbf{let}_{\mathsf{int}\to\mathsf{int}}\ \mathtt{f} = \mathbf{fun}(\mathtt{n} : \mathsf{int}) \mapsto \{...\};$
`const int x = 3;`	$\longleftrightarrow$	$\mathbf{let}_{\mathsf{int}}\ \mathtt{x} = 3;$
`f(x);`	$\longleftrightarrow$	$\mathtt{f}(\mathtt{x});$
`int y;`	$\longleftrightarrow$	$\mathbf{let}_{\mathsf{ptr(int)}}\ \mathtt{y} = \mathsf{stackCell}_{\mathsf{int}}();$
`f(y);`	$\longleftrightarrow$	$\mathtt{f}(\mathsf{get}(\mathtt{y}));$
`int* const z = malloc(sizeof(int));`	$\longleftrightarrow$	$\mathbf{let}_{\mathsf{ptr(int)}}\ \mathtt{z} = \mathsf{heapCell}_{\mathsf{int}}();$
`f(*z);`	$\longleftrightarrow$	$\mathtt{f}(\mathsf{get}(\mathtt{z}));$
`free(z);`	$\longleftrightarrow$	$\mathsf{free}(\mathtt{z});$