Script generated by TTT

Title: Seidl: Programmoptimierung (10.12.2012)
Date: Mon Dec 10 15:05:26 CET 2012
Duration: 88:13 min
Pages: 44

2. Idea: Elimination of Tail Recursion

```c
f () {  int b;
    if (a2 <= 1) {  ret = a1; goto _exit; }
    b = a1 * a2;
    a2 = a2 - 1;
    a1 = b;
    f();
    _exit :
}
```

After the procedure call, nothing in the body remains to be done.

We may directly jump to the beginning.

... after having reset the locals to 0.

... this yields in the Example:

```c
f () {  int b = 0; }
_f :  if (a2 <= 1) {  ret = a1; goto _exit; }
    b = a1 * a2;
    a2 = a2 - 1;
    a1 = b;
    goto _f;
    _exit :
}
```

// It works, since we have ruled out references to variables!

Transformation 11:
Warning:

→ This optimization is crucial for programming languages without iteration constructs !!!
→ Duplication of code is not necessary    :-) 
→ No variable renaming is necessary     :-) 
→ The optimization may also be profitable for non-recursive tail calls    :-) 
→ The corresponding code may contain jumps from the body of one procedure into the body of another ???

Background 4:    Interprocedural Analysis

So far, we can analyze each procedure separately.

→ The costs are moderate    :-) 
→ The methods also work in presence of separate compilation   :-) 
→ At procedure calls, we must assume the worst case    :( 
→ Constant propagation only works for local constants    :-(

Question:

How can recursive programs be analyzed ???

Example:    Constant Propagation

```
main() { int t;
    0;
    if (t) M[17] = 3;
    f = t;
    work();
    ret = 1 - ret;
    }
```

```
work() { if (a) work();
    ret = a;
    }
```

```
Example:    Constant Propagation
```
```
main() 0
    t = 0;
    Neg (t)
    M[17] = 3;
    1
    work();
    3
    a = t;
    4
    work();
    5
    ret = 1 - ret;
```
```
work ()
    7
    Neg (a)
    Pos (a)
    8
    work();
    10
    ret = a;
```

```
Neg (t)
    1
    Pos (t)
    2
    M[17] = 3;
    3
    a = t;
    4
    work();
    5
    ret = 1 - ret;
```
(1) Functional Approach:

Let $\mathcal{D}$ denote a complete lattice of (abstract) states.

Idea:

Represent the effect of $f()$ by a function:

$$[f]^f : \mathcal{D} \to \mathcal{D}$$
In order to determine the effect of a call edge \( k = (n, f(); v) \) we require abstract functions:

\[
\begin{align*}
\text{enter}^t & : \mathbb{D} \rightarrow \mathbb{D} \\
\text{combine}^t & : \mathbb{D}^2 \rightarrow \mathbb{D}
\end{align*}
\]

Then we define:

\[
[k]^t D = \text{combine}^t(D, [f]^t(\text{enter}^t D))
\]

---

(1) Functional Approach:

Let \( \mathbb{D} \) denote a complete lattice of (abstract) states.

Idea:

Represent the effect of \( f() \) by a function:

\[
[f]^t : \mathbb{D} \rightarrow \mathbb{D}
\]

---

... for Constant Propagation:

\[
\begin{align*}
\mathbb{D} & = (\text{Vars} \rightarrow \mathbb{Z})_\bot \\
\text{enter}^t D & = \begin{cases} \\
\bot & \text{if } D = \bot \\
D|\text{Globals} \oplus \{x \rightarrow 0 \mid x \in \text{Locals}\} & \text{otherwise} \\
\end{cases} \\
\text{combine}^t(D_1, D_2) & = \begin{cases} \\
\bot & \text{if } D_1 = \bot \lor D_2 = \bot \\
D_1|\text{Locals} \oplus D_2|\text{Globals} & \text{otherwise} \\
\end{cases}
\end{align*}
\]
The effects \([/f]^2\) then can be determined by a system of constraints over the complete lattice \(\mathcal{D} \rightarrow \mathcal{D}\):

\[
[\nu]^2 \supseteq \text{ld} & \quad & \nu \quad \text{entry point} \\
[\nu]^2 \supseteq [k]^2 \circ [\nu]^2 & \quad & k = (u, \_, \nu) \quad \text{edge} \\
[/f]^2 \supseteq \text{stop}_f^2 & \quad & \text{stop}_f \quad \text{end point of } \ f
\]

\([\nu]^2 : \mathcal{D} \rightarrow \mathcal{D}\) describes the effect of all prefixes of computation forests \(w\) of a procedure which lead from the entry point to \(\nu\).

**Problems:**

- How can we represent functions \(f : \mathcal{D} \rightarrow \mathcal{D}\)?
- If \(\#\mathcal{D} = \infty\), then \(\mathcal{D} \rightarrow \mathcal{D}\) has infinite strictly increasing chains.

**Simplification:** Copy-Constants

\[
\rightarrow \quad \text{Conditions are interpreted as } \vdash : \quad \text{\(\therefore\)}
\]

\[
\rightarrow \quad \text{Only assignments } x = e; \quad \text{with } e \in Vars \cup \mathbb{Z} \quad \text{are treated exactly } \vdash : \quad \text{\(\therefore\)}
\]
Problems:

- How can we represent functions \( f : D \rightarrow D \)???
- If \( \#D = \infty \), then \( D \rightarrow D \) has infinite strictly increasing chains :-(

Simplification: Copy-Constants

- Conditions are interpreted as : :-)  
- Only assignments \( x = e ; \) with \( e \in Vars \cup \mathbb{Z} \) are treated exactly :-(

Observation:

- The effects of assignments are:
  \[
  [x = c]^D D = \begin{cases} 
  D \oplus \{ x \mapsto c \} & \text{if } c = e \in \mathbb{Z} \\
  D \oplus \{ x \mapsto (D,y) \} & \text{if } e = y \in Vars \\
  D \oplus \{ x \mapsto \top \} & \text{otherwise}
  \end{cases}
  \]

- Let \( \forall V \) denote the (finite !!!) set of constant right-hand sides. Then variables may only take values from \( \forall^\top :-) \)
- The occurring effects can be taken from \( D_f \rightarrow D_f \) with \( D_f = (Vars \rightarrow \forall^\top)_\perp \)
- The complete lattice is huge, but finite !!!
Observation:

\[
[x = c]^D = \begin{cases}
  D \oplus \{ x \mapsto c \} & \text{if } c = c \in \mathbb{Z} \\
  D \oplus \{ x \mapsto y \} & \text{if } c \neq c \in \text{Vars} \\
  D \oplus \{ x \mapsto T \} & \text{otherwise}
\end{cases}
\]

→ The effects of assignments are:

→ Let \( \mathcal{V} \) denote the (finite) set of constant right-hand sides. Then variables may only take values from \( \mathcal{V} \).

→ The occurring effects can be taken from \( \mathcal{D}_f \) as \( \mathcal{D}_f = (\text{Vars} \to \mathcal{V}) \).

→ The complete lattice is huge, but finite

---

Observation:

\[
\frac{\mathcal{D}_f}{\mathcal{D}_f} = (c \land \neg c) + \neg c
\]

→ The effects of assignments are:

\[
[x = c]^D = \begin{cases}
  D \oplus \{ x \mapsto c \} & \text{if } c = c \in \mathbb{Z} \\
  D \oplus \{ x \mapsto (D \land y) \} & \text{if } c \neq y \in \text{Vars} \\
  D \oplus \{ x \mapsto T \} & \text{otherwise}
\end{cases}
\]

→ Let \( \mathcal{V} \) denote the (finite) set of constant right-hand sides. Then variables may only take values from \( \mathcal{V} \).

→ The occurring effects can be taken from \( \mathcal{D}_f \) as \( \mathcal{D}_f = (\text{Vars} \to \mathcal{V}) \).

→ The complete lattice is huge, but finite

---

Improvement:

\[
\begin{align*}
  \mathcal{X}_1 &\geq \sum_{i=1}^{\# \text{Vars}} \mathcal{X}_i \\
  \mathcal{X} &\to \sum_{i=1}^{\# \text{Vars}} \mathcal{X}_i
\end{align*}
\]

→ Not all functions from \( \mathcal{D}_f \to \mathcal{D}_f \) will occur.

→ All occurring functions \( \lambda \mathcal{D}, \perp \neq \mathcal{M} \) are of the form:

\[
\mathcal{M} = \{ x \mapsto (b \cup \bigcup_{y \in \mathcal{U}} y) \mid x \in \text{Vars} \} \quad \text{where:}
\]

\[
\mathcal{M} \mathcal{D} = \{ x \mapsto (b \cup \bigcup_{y \in \mathcal{U}} D y) \mid x \in \text{Vars} \} \quad \text{for } \mathcal{D} \neq \perp
\]

→ Let \( \mathcal{M} \) denote the set of all these functions. Then for \( \mathcal{M}_1, \mathcal{M}_2 \in \mathcal{M} \) \( (\mathcal{M}_1 \neq \lambda \mathcal{D}, \perp \neq \mathcal{M}_2) \):

\[
(\mathcal{M}_1 \cup \mathcal{M}_2) \mathcal{X} = (\mathcal{M}_1 \mathcal{X}) \cup (\mathcal{M}_2 \mathcal{X})
\]

→ For \( \# \text{Vars} = \mathcal{M} \) has height \( \mathcal{O}(k^2) \)
Improvement:

→ Not all functions from $\mathcal{D}_f \rightarrow \mathcal{D}_f$ will occur

→ All occurring functions $\lambda D, \perp \neq M$ are of the form:

$M = \{ x \mapsto (b_x \cup \bigcup_{y \in y} y) | x \in \text{Vars} \}$ where:

$MD = \{ x \mapsto (b_x \cup \bigcup_{y \in y} D y) | x \in \text{Vars} \}$ for $D \neq \perp$

→ Let $\mathbb{M}$ denote the set of all these functions. Then for $M_1, M_2 \in \mathbb{M}$ $(M_1 \not= \lambda D, \perp \neq M_2)$:

$(M_1 \cup M_2) x = (M_1 x) \cup (M_2 x)$

→ For $k = \# \text{Vars}$, $M$ has height $O(k^2)$

\[ \text{Improvement (Cont.):} \]

→ Also, composition can be directly implemented:

$(M_1 \circ M_2) x = b' \cup \bigcup_{y \in y} b_y$ with

$b' = b \cup \bigcup_{y \in y} b_y$

$I' = \bigcup_{y \in y} I_y$ where

$M_1 x = b \cup \bigcup_{y \in y} y$

$M_2 z = b_z \cup \bigcup_{y \in y} z$

→ The effects of assignments then are:

$[x = c]^{(M_1 \circ M_2)} = \begin{cases} 
\text{ld}_\text{Vars} \oplus \{ x \rightarrow c \} & \text{if } e = c \in \mathbb{Z} \\
\text{ld}_\text{Vars} \oplus \{ x \rightarrow y \} & \text{if } e = y \in \text{Vars} \\
\text{ld}_\text{Vars} \oplus \{ x \rightarrow T \} & \text{otherwise}
\end{cases}$
... in the Example:

\[
[t = 0]^f = \{ a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto 0 \}
\]

\[
[a_1 = t]^f = \{ a_1 \mapsto t, \text{ret} \mapsto \text{ret}, t \mapsto t \}
\]

In order to implement the analysis, we additionally must construct the effect of a call \( k = (\_, f(\_), \_) \) from the effect of a procedure \( f \):

\[
[k]^f = H ([f]^f)
\]

where:

\[
H (M) = \text{id}_{\text{Locals}} \oplus (M \circ \text{enter}^4)|_{\text{globals}}
\]

\[
\text{enter}^4 x = \begin{cases} 
  x & \text{if } x \in \text{Globals} \\
  0 & \text{otherwise}
\end{cases}
\]

... in the Example:

If \([\text{work}]^f = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}\) then \(H[\text{work}]^f = \text{id}_{[t]} \oplus \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1 \}\)

\[
= \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \}
\]

Now we can perform fixpoint iteration :-)

... in the Example:

\[
[t = 0]^f = \{ a_1 \mapsto a_1, \text{ret} \mapsto \text{ret}, t \mapsto 0 \}
\]

\[
[a_1 = t]^f = \{ a_1 \mapsto t, \text{ret} \mapsto \text{ret}, t \mapsto t \}
\]

In order to implement the analysis, we additionally must construct the effect of a call \( k = (\_, f(\_), \_) \) from the effect of a procedure \( f \):

\[
[k]^f = H ([f]^f)
\]

where:

\[
H (M) = \text{id}_{\text{Locals}} \oplus (M \circ \text{enter}^4)|_{\text{globals}}
\]

\[
\text{enter}^4 x = \begin{cases} 
  x & \text{if } x \in \text{Globals} \\
  0 & \text{otherwise}
\end{cases}
\]
... in the Example:

If \[ \text{work} \] then \[ H \text{[work]} = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \]

Now we can perform fixpoint iteration.

\[ \left( (8, \ldots, 9) \circ [8] \right)^\omega = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \circ \]
\[ = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \]

\[ \left( (8, \ldots, 9) \circ [8] \right)^\omega = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \circ \]
\[ = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \]

\[ \left( (8, \ldots, 9) \circ [8] \right)^\omega = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \circ \]
\[ = \{ a_1 \mapsto a_1, \text{ret} \mapsto a_1, t \mapsto t \} \]
If we know the effects of procedure calls, we can put up a constraint system for determining the abstract state when reaching a program point:

\[ \mathcal{R}[\text{main}] \sqsubseteq \text{enter}^4_d \]
\[ \mathcal{R}[f] \sqsubseteq \text{enter}^2(\mathcal{R}[u]) \quad k = (u, f(); \text{...}) \quad \text{call} \]
\[ \mathcal{R}[v] \sqsubseteq \mathcal{R}[f] \quad v \quad \text{entry point of } f \]
\[ \mathcal{R}[v] \sqsubseteq [k]^2(\mathcal{R}[u]) \quad k = (u, v) \quad \text{edge} \]

Discussion:

- At least copy-constants can be determined interprocedurally.
- For that, we had to ignore conditions and complex assignments :-(
- In the second phase, however, we could have been more precise :-)
- The extra abstractions were necessary for two reasons:
  1. The set of occurring transformers \( M \subseteq D \rightarrow D \) must be finite;
  2. The functions \( M \in M \) must be efficiently implementable :)
- The second condition can, sometimes, be abandoned ...
Discussion:

- At least copy-constants can be determined interprocedurally.
- For that, we had to ignore conditions and complex assignments :(.
- In the second phase, however, we could have been more precise :-)
- The extra abstractions were necessary for two reasons:
  
  (1) The set of occurring transformers \( M \subseteq D \rightarrow D \) must be finite;
  
  (2) The functions \( M \in M \) must be efficiently implementable ;-

- The second condition can, sometimes, be abandoned ...