A High-Level IR Transformation System

Herbert Jordan,
Peter Thoman, and
Thomas Fahringer
University of Innsbruck

Insieme

Establish an **integrated**

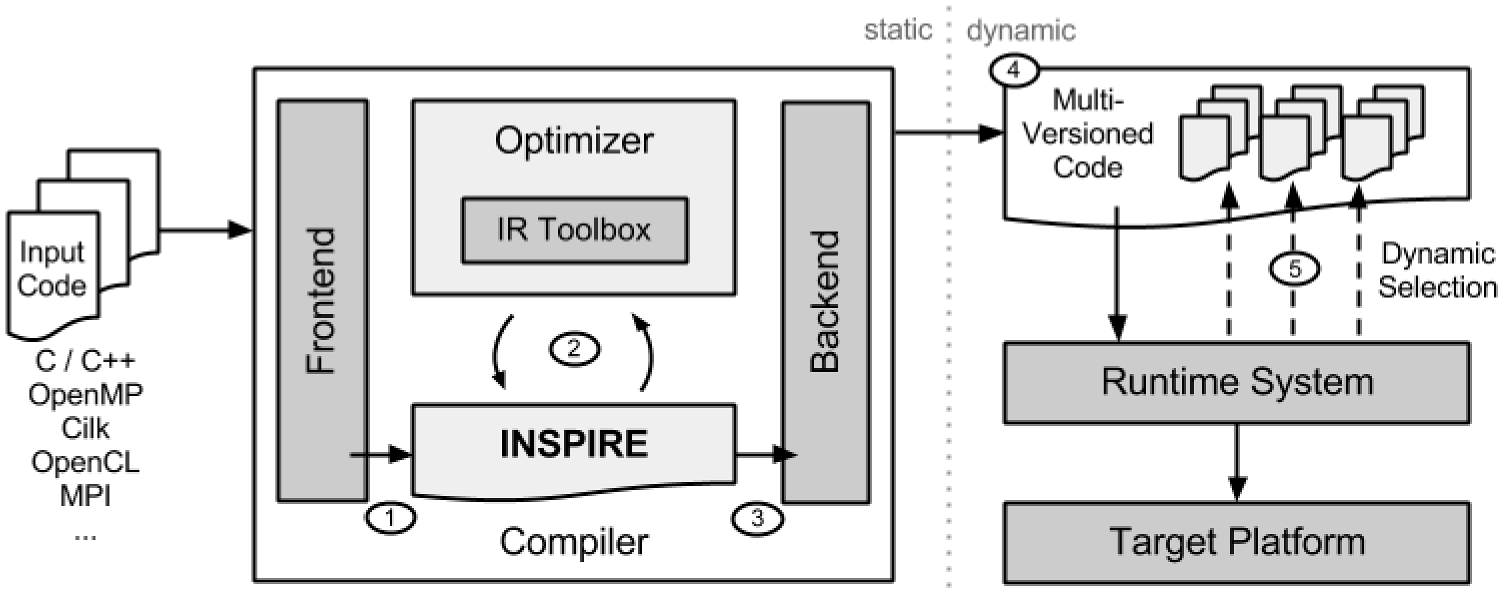
Compiler & Runtime Research **Platform**

for

analyzing / manipulating / (auto-)tuning

parallel C/C++ applications

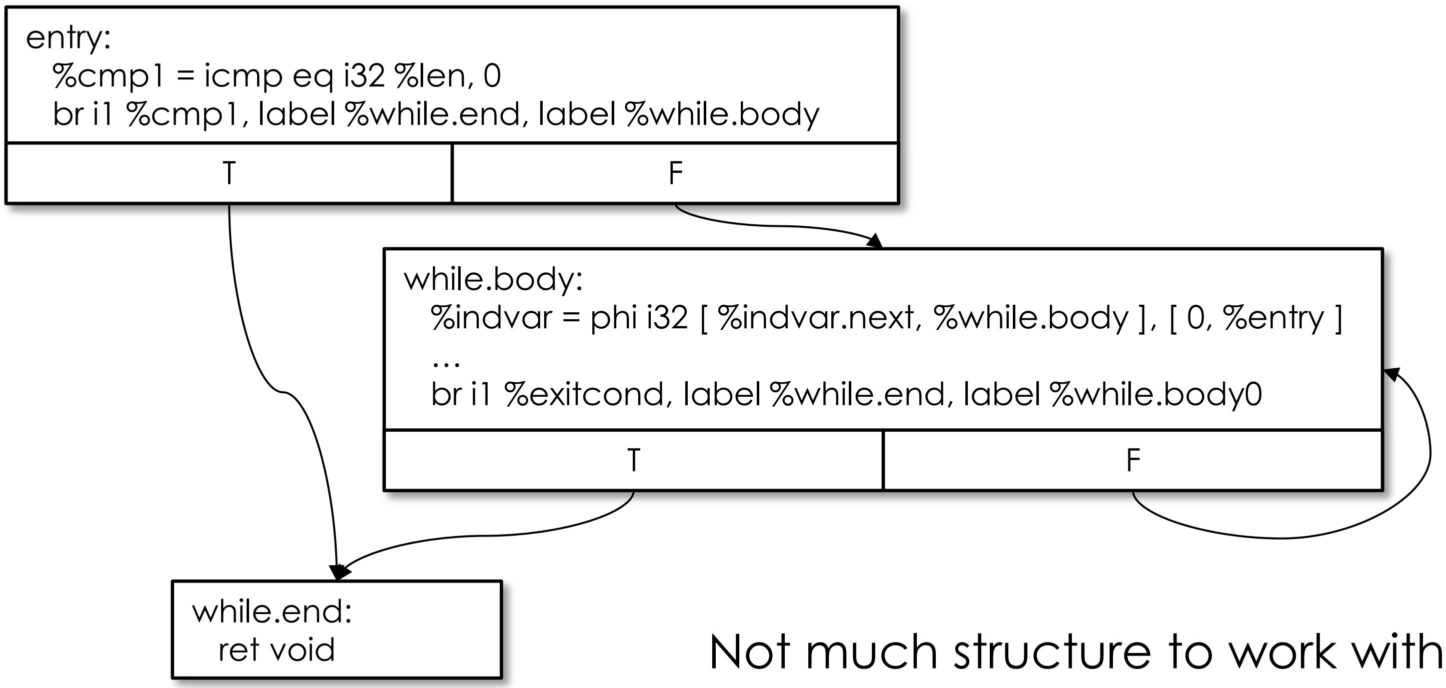
The Insieme Infrastructure



developed @ University of Innsbruck

Code Transformations

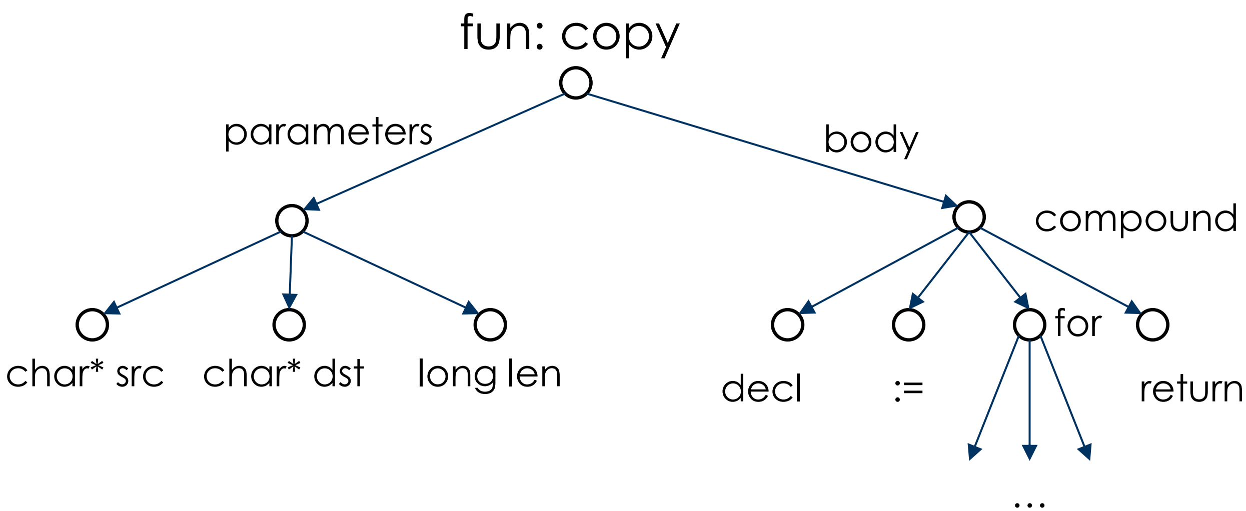
- Traditional Compiler (GCC, LLVM):
  - Low-level IRs



Not much structure to work with

Code Transformations (2)

- Source-to-Source Compiler
  - High-level IR – Rose, Clang, CIL, **INSPIRE**



Much more (high-level) structure

How to transform ASTs?

- Typically: **hand-coded** manipulations
  1. find target
  2. collect input pieces
  3. distinguish cases
  4. synthesize replacement
  5. integrate replacement

```
NodeMap publicToPrivateMap;
NodeMap privateToPublicMap;
// implement private copies where required
for_each(allp, [&](const ExpressionPtr& varExp) {
    const auto& expType = varExp->getType();
    VariablePtr pVar = build.variable(expType);
    publicToPrivateMap[varExp] = pVar;
    privateToPublicMap[pVar] = varExp;
    DeclarationStmtPtr decl = build.declarationStmt(pVar, build.undefine
    if(contains(firstPrivates, varExp)) {
        // make sure to actually get *copies* for firstprivate initializ
        if(core::analysis::isRefType(expType)) {
            VariablePtr fpPassVar = build.variable(core::analysis::getRe
            DeclarationStmtPtr fpPassDecl = build.declarationStmt(fpPass
            outsideDecls.push_back(fpPassDecl);
            decl = build.declarationStmt(pVar, build.refVar(fpPassVar));
        }
        else {
            decl = build.declarationStmt(pVar, varExp);
        }
    }
    if(clause->hasReduction() && contains(clause->getReduction().getVars
        decl = build.declarationStmt(pVar, getReductionInitializer(claus
    }
    replacements.push_back(decl);
});
// implement copyin for threadprivate vars
if(parallelP && parallelP->hasCopyin()) {
    for(const ExpressionPtr& varExp : parallelP->getCopyin()) {
        // assign master copy to private copy
        StatementPtr assignment = build.assign(
            static_pointer_cast<const Expression>(handleThreadprivate(va
            build.deref(static_pointer_cast<const Expression>(handleThre
        replacements.push_back(assignment);
    }
}
```

- Result:
  - labor intensive
  - error prone
  - reduced maintainability

Structured Approaches

- **ASTs** are 'somewhat' similar to **Terms**
  => use term rewriting – e.g. **Stratego** or **TXL**

- Transformations:
  - set of "**pattern => replacement**" rules
  - input is transformed by applying rules

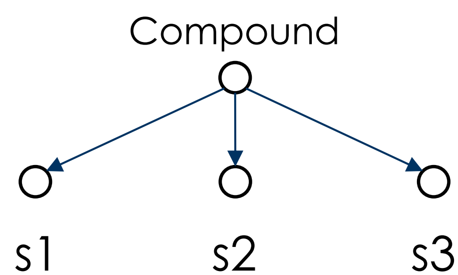
- Problem:
  - external system, not directly adaptable
  - ASTs are just '**somewhat**' similar to Terms

ASTs vs. Terms

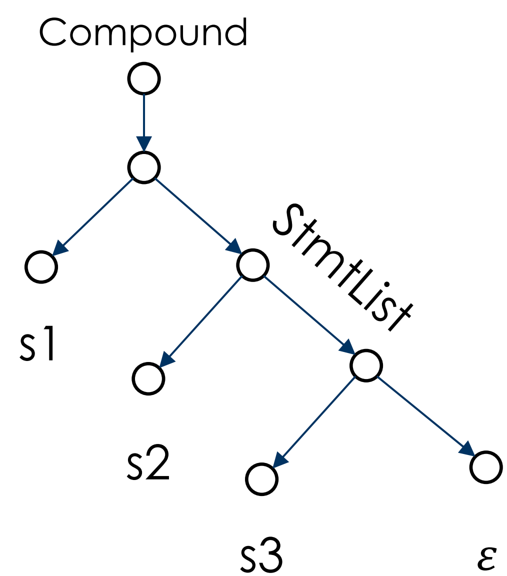
- **Terms** are isomorph to **Algebraic Structures**
  - every symbol has **fixed arity**

```
{
  s1;
  s2;
  s3;
}
```

Source



Desired AST



Algebraic Structure

Our Objective

- Design a Transformation System that is
  - **declarative**
  - operating on **arbitrary trees**
    - in particular High-Level Compiler IRs
  - supporting **deep inspection**
    - beyond flat pattern matching

Basic Setup

- Tree Structure:

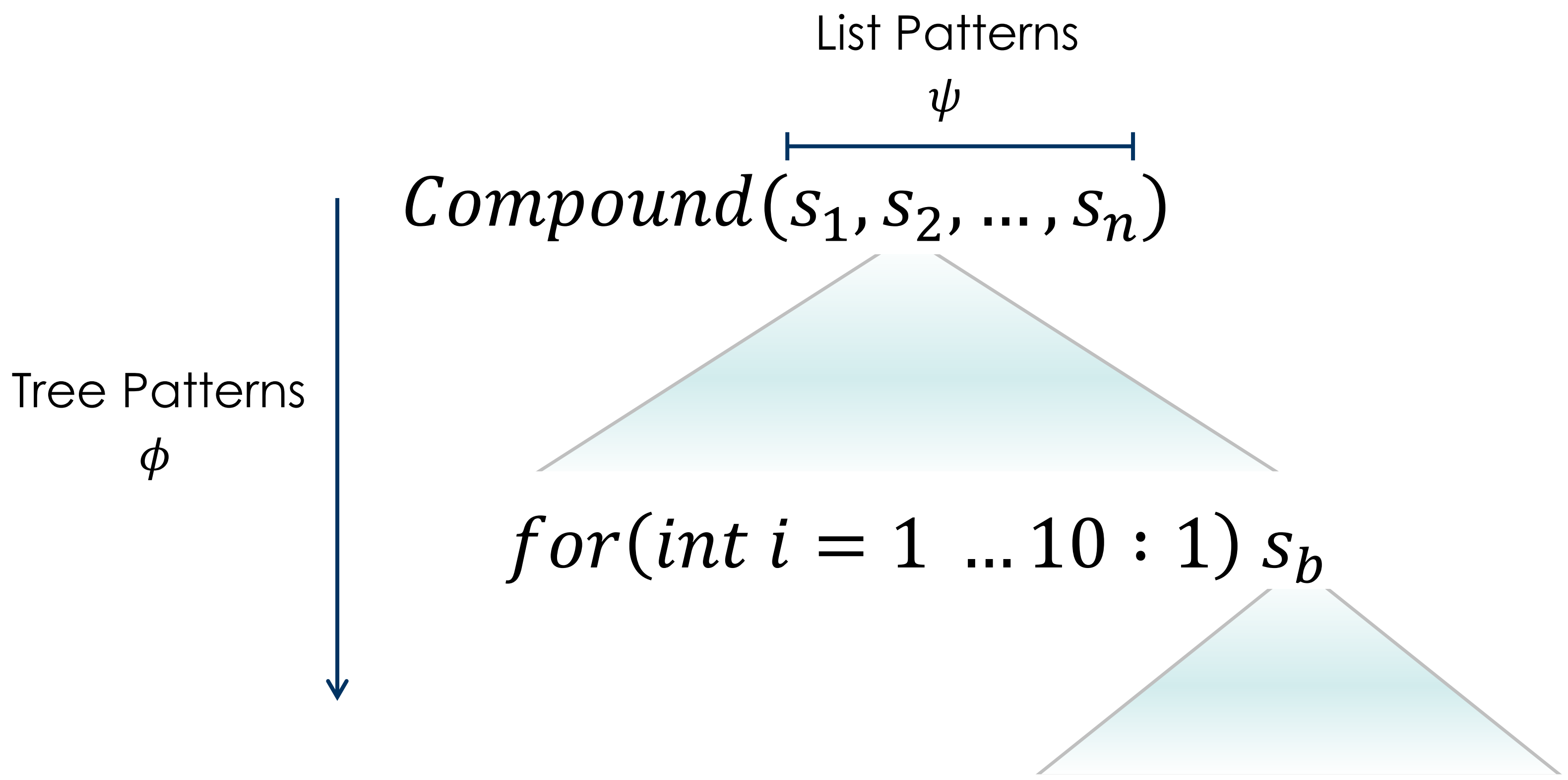
$$T ::= a \mid k(T^*)$$

- Rule structure:

$$\phi \rightarrow \tau$$

ϕ … is a tree pattern
τ … is a tree generator

Patterns - Concept



Pattern

- ***Tree Patterns*** – matching trees

$$\phi ::= \_ \mid t \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \_(\psi) \mid k(\psi) \mid$$
$$\mid x : \phi \mid aT(\phi) \mid rT.x(\phi) \mid rec.x$$

- ***List Patterns*** – matching forests

$$\psi ::= \epsilon \mid \phi \mid \psi, \psi \mid \psi \vee \psi \mid x : \psi \mid \psi^*$$

Generators

- Tree Generators

$$\tau ::= \upsilon \mid k\,(\sigma) \;\mid \tau\,[\tau/\tau]$$

- List Generators

$$\sigma ::= \upsilon \mid \epsilon \mid \;[\tau] \;\mid \sigma, \sigma$$

- Value Generators

$$\upsilon ::= \lambda_c \mid \lambda_t\,(\upsilon) \;\mid \tau \mid \sigma \mid$$
$$\mid let\ x = \upsilon\ in\ \upsilon \mid \forall x \in \upsilon\ .\ \upsilon$$

Semantic – Tree Patterns

$t, m, n, r \vdash \_$	iff true	(wildcard)
$t, m, n, r \vdash t$	iff $t = t$	(constant)
$t, m, n, r \vdash \neg\phi$	iff not $t, m', n', r \vdash \phi$ and $m \subseteq m'$ and $n \subseteq n'$	(negation)
$t, m, n, r \vdash \phi_1 \wedge \phi_2$	iff $t, m', n', r \vdash \phi_1$ and $t, m, n, r \vdash \phi_2$ and $m' \subseteq m$ and $n' \subseteq n$	(and)
$t, m, n, r \vdash \phi_1 \vee \phi_2$	iff $t, m, n, r \vdash \phi_1$ or $t, m, n, r \vdash \phi_2$	(or)
$t, m, n, r \vdash \_(\psi)$	iff $t = k(t_1, \ldots, t_l)$ and $[t_1, \ldots, t_l], m, n, r \vdash \psi$	(any node)
$t, m, n, r \vdash k(\psi)$	iff $t = k(t_1, \ldots, t_l)$ and $[t_1, \ldots, t_l], m, n, r \vdash \psi$	(node)
$t, m, n, r \vdash x : \phi$	iff $t, m \setminus \{x\}, n, r \vdash \phi$ and $(x \mapsto t) \in m$	(var)
$t, m, n, r \vdash aT(\phi)$	iff t' subtree of t and $t', m, n, r \vdash \phi$	(any tree)
$t, m, n, r \vdash rT.x(\phi)$	iff $t, m, n, \{x \mapsto (\phi, m, n)\} \oplus r \vdash \phi$	(recursion)
$t, m, n, r \vdash rec.x$	iff $x \mapsto (\phi, m', n') \in r$ and $t, m', n', r \vdash \phi$	(rec. end)

Pattern Examples

- *Task:*

  is variable v referenced within some code fragment?

- *Pattern:*

$$aT(v)$$

List Pattern

- *Task:*

  is expression exp a full expression within a given compound statement?

- *Pattern:*

$$\{\_^*, exp, \_^*\}$$

Variables

- *Task:*

  Get IR variable declared by
  a IR variable declaration

- *Pattern:*

$$decl(\$x)$$

matched against "$\mathrm{int}\ v12$" it yields $\{x = v12\}$

Variables

- *Task:*

  Get all variables declared in
  a compound statement

- *Pattern:*

$$\{(\neg decl(\_))^*, (decl(\$x), (\neg decl(\_))^*)^*\}$$

matched against

$$\{int\ a = 5; f(a); bool\ b = true; int\ c = 7;\}$$

it yields $\{x = [a, b, c]\}$

Variable Binding

- *Task:*

  Check whether a declared variable is never used.

- *Pattern:*

$$\{decl(\$x), (\neg aT(\$x))^*\}$$

Once $\$x$ is bound in outer scope, inner is fixed!

Recursive Patterns

- *Task:*

  Collect all loops within
  a for-loop nest

- *Pattern:*

$$rT.x(\$l : for(rec.x \wedge \neg for(\_))$$

Variable $\$l$ is collecting a list of for-loops.

Implementation

- Implemented within the **Insieme** Compiler
  - templated **utility library** (C++11)
  - matching algorithm:
    - **recursive back-tracking** + pruning heuristics
  - **Overloaded operators** for composing patterns and generators (extendable)

Real-World Transformation

- Eliminate redundant **sync** calls (Cilk)

```
{
    {
        spawn f(a);
        spawn f(b);
        ...;
        sync;
    }
    ...;
    sync;        redundant
}
```

Identify Redundant Syncs

In C++11 notation:

```
auto unsynced = rT(spawn | node(*any << aT(rec) << *!sync));
auto synced = ! unsynced;

auto p = compound(
      *synced << var("x", sync) << *any
);
```

… and the rest:

Create a tree generator expression:

```
auto r = substitute(root, var("x"), noop);
```

Create a rule:

```
Rule syncElimination = Rule( p, r );
```

Apply the rule:

```
auto out = syncElimination( in );
```

Complete Example

```
auto synced = ! rT(spawn | node(*any << aT(rec) << *!sync));
auto p = compound(
      *synced << var("x", sync) << *any
);

auto r = substitute(root, var("x"), noop);
Rule syncElimination = Rule( p, r );

auto out = syncElimination( in );
```

Conclusion

- Our solution provides a **descriptive** infrastructure for tree **transformations**
- patterns = **unification** + **regex**
  + any-where-in-tree primitive (**aT**)
  + recursive tree primitive (**rT**)
- Generic C++ implementation
  - **portable** to other domains (trees)
  - support for **domain-specific primitives**

Thank You!

Visit: http://insieme-compiler.org

Contact: herbert@dps.uibk.ac.at