Concurrent Erlang Flow Graphs

Manfred Widera

Fachbereich Informatik, FernUniversität in Hagen
D-58084 Hagen, Germany
Manfred.Widera@fernuni-hagen.de

Erlang User Conference 2005

General Notes of Software Testing

Software is usually tested in several stages.

1. Component testing: early testing of individual functions or modules
2. Integration testing: testing of interfaces and interaction of several modules
3. System testing: testing of the whole system (stress tests, beta tests, . . .)

General Notes of Software Testing

Software is usually tested in several stages.

1. Component testing: early testing of individual functions or modules
2. Integration testing: testing of interfaces and interaction of several modules
3. System testing: testing of the whole system (stress tests, beta tests, . . .)

General Notes of Software Testing

Software is usually tested in several stages.

1. Component testing: early testing of individual functions or modules
2. Integration testing: testing of interfaces and interaction of several modules
3. System testing: testing of the whole system (stress tests, beta tests, . . .)

General Notes of Software Testing

Software is usually tested in several stages.

1. Component testing: early testing of individual functions or modules
2. Integration testing: testing of interfaces and interaction of several modules
3. System testing: testing of the whole system (stress tests, beta tests, . . .)

General Notes of Software Testing

Software is usually tested in several stages.

1. Component testing: early testing of individual functions or modules
2. Integration testing: testing of interfaces and interaction of several modules
3. System testing: testing of the whole system (stress tests, beta tests, . . .)

Flow graph oriented test methods just apply for component testing.

Outline

Definition and Generation of Flow Graphs

Data Flow Analysis

Computation of Edges in Flow Graphs

Test Case Analysis

Conclusion

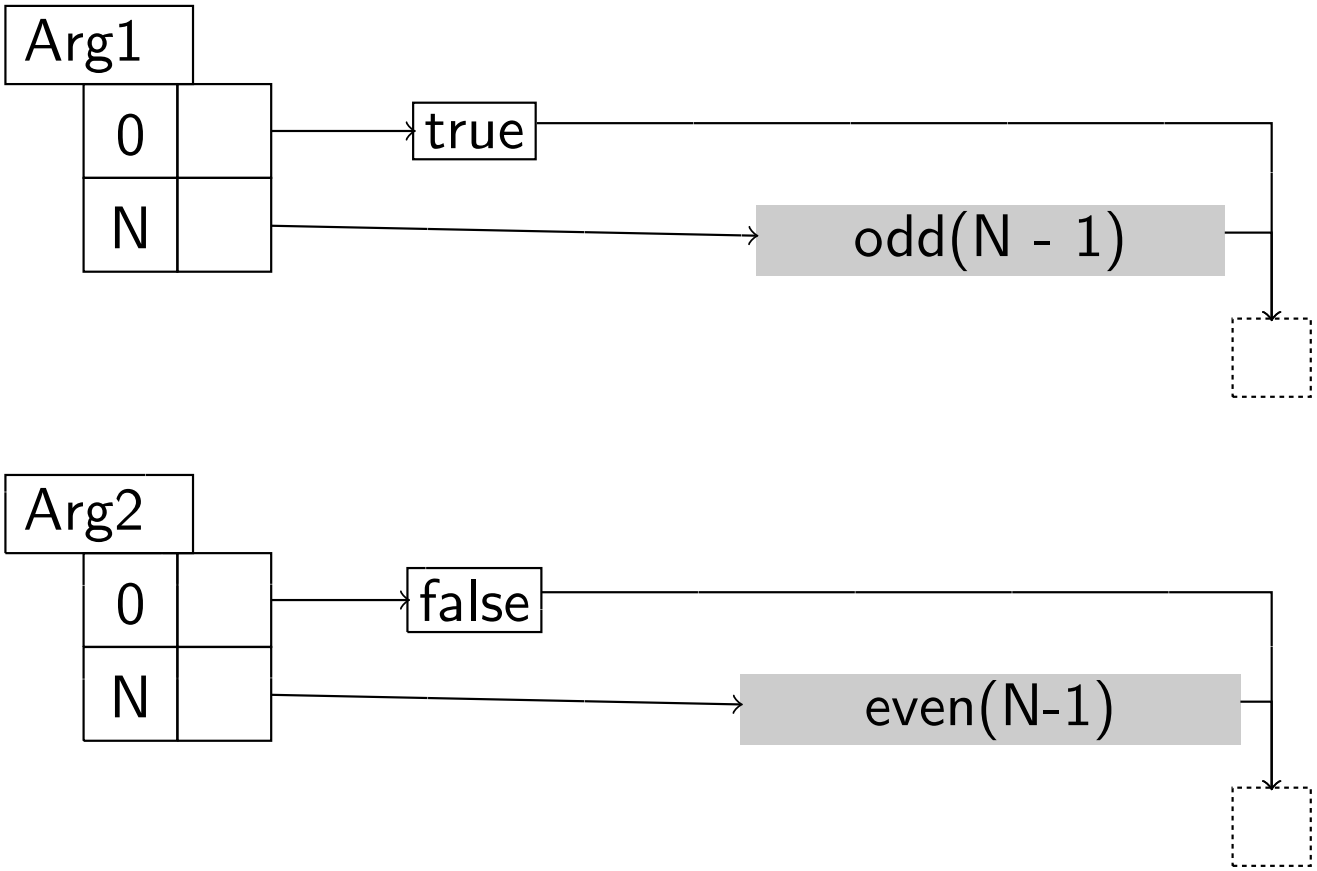
Definition and Generation of Flow Graphs

```
even(0) -> true;
even(N) -> odd(N - 1).

odd(0) -> false;
odd(N) -> even(N - 1).
```

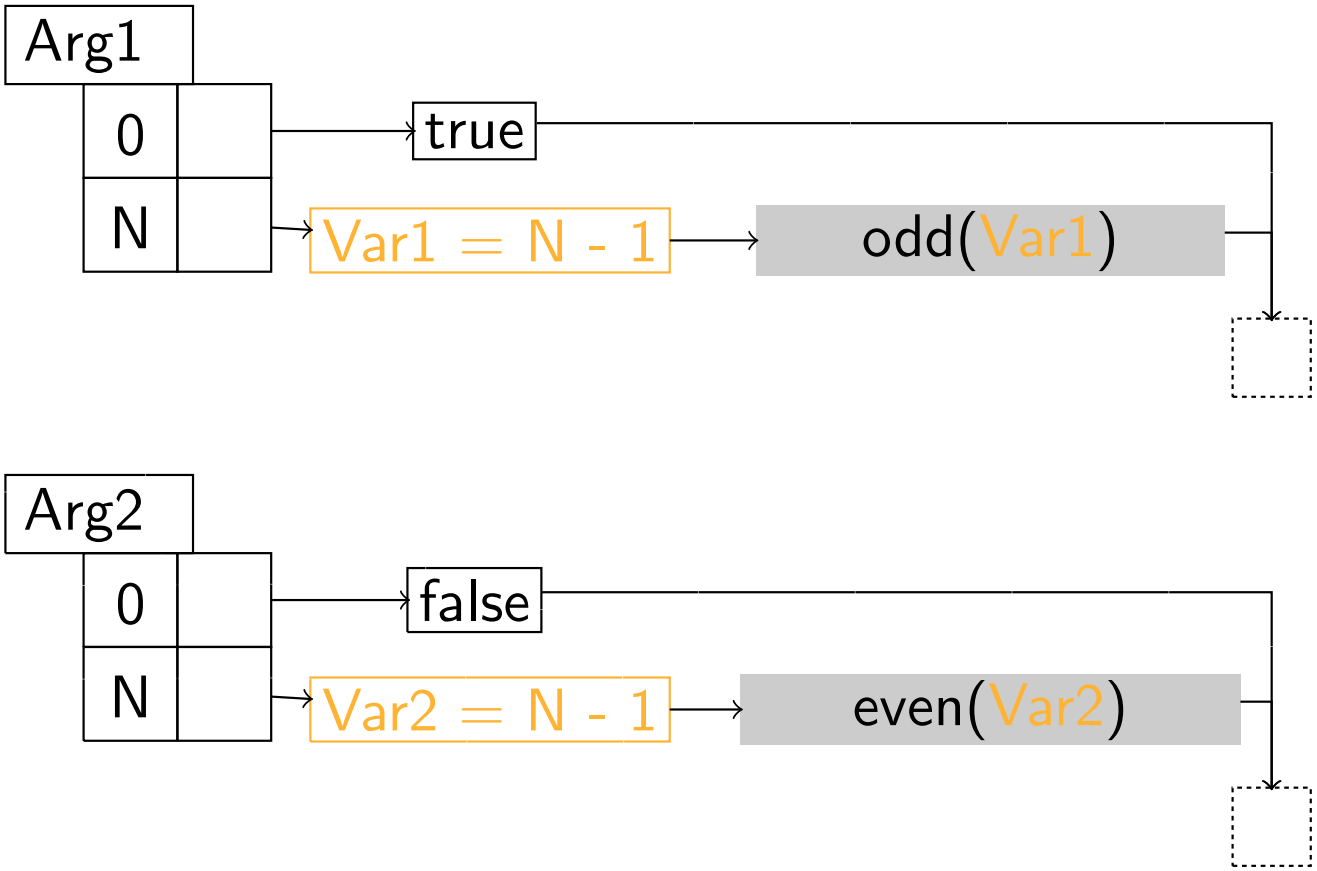
One or several modules are read.

Definition and Generation of Flow Graphs



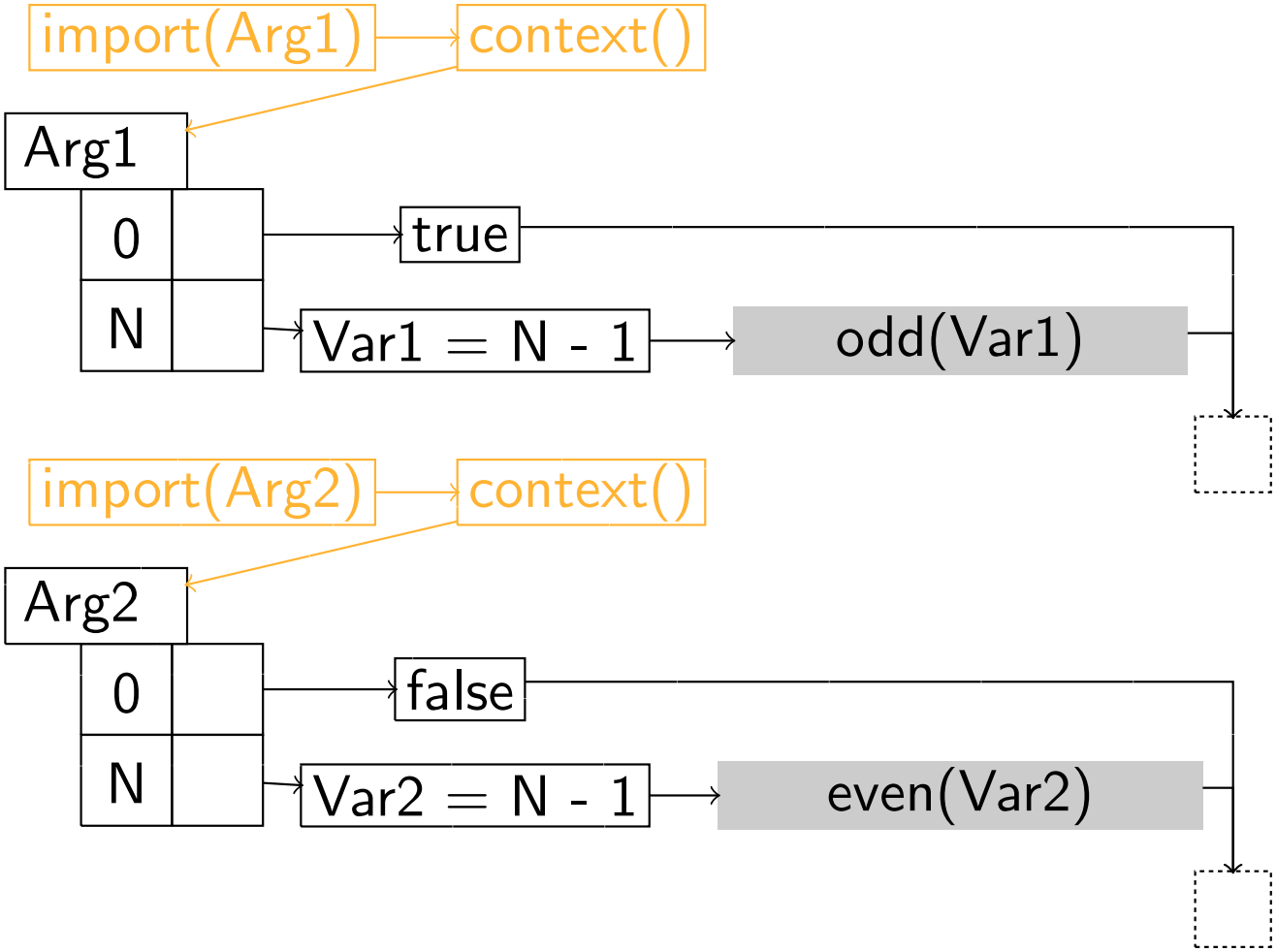
Individual expressions are transformed into flow graph representation.

Definition and Generation of Flow Graphs



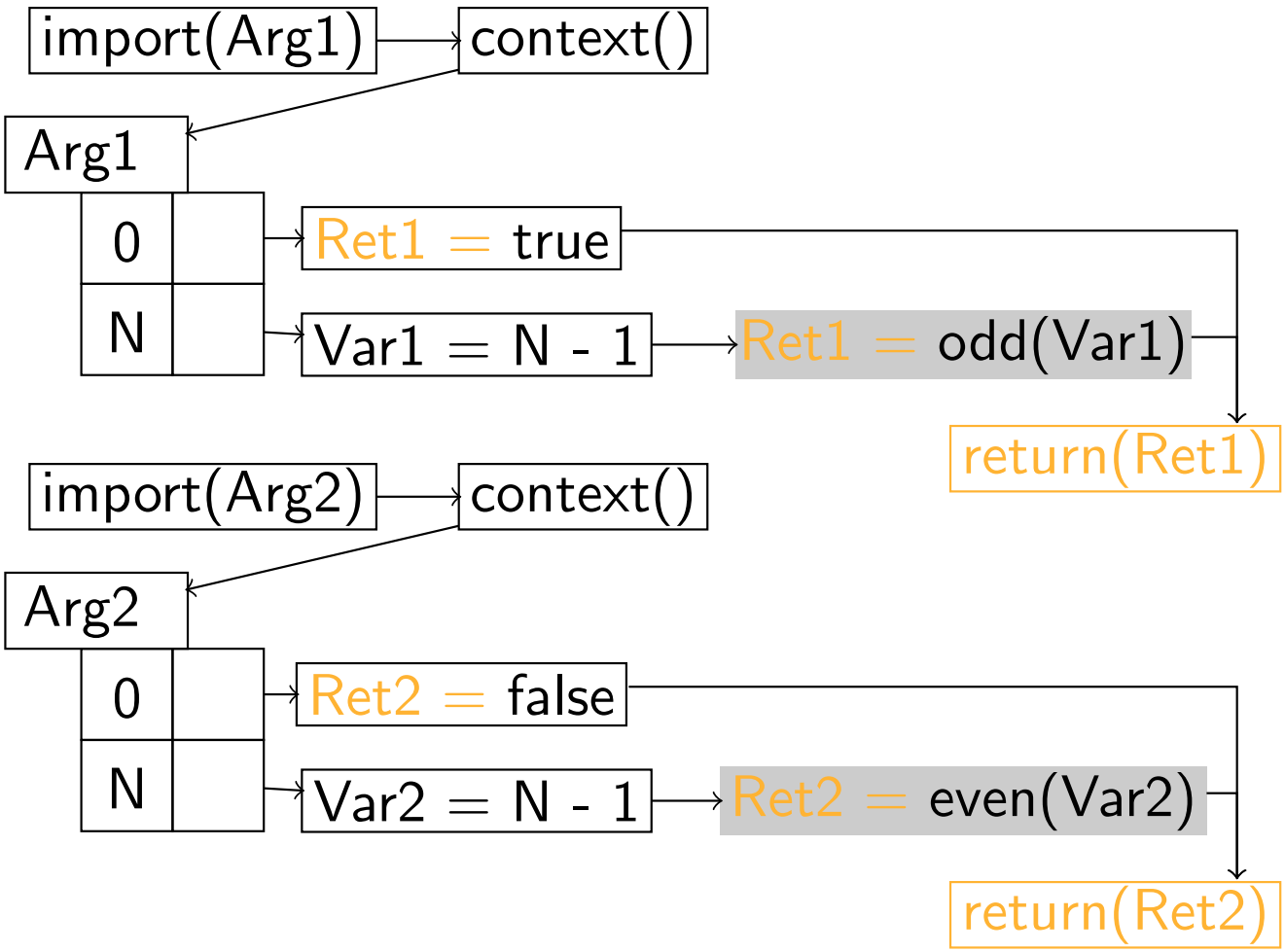
Fresh variables are introduced for intermediate computation results.

Definition and Generation of Flow Graphs



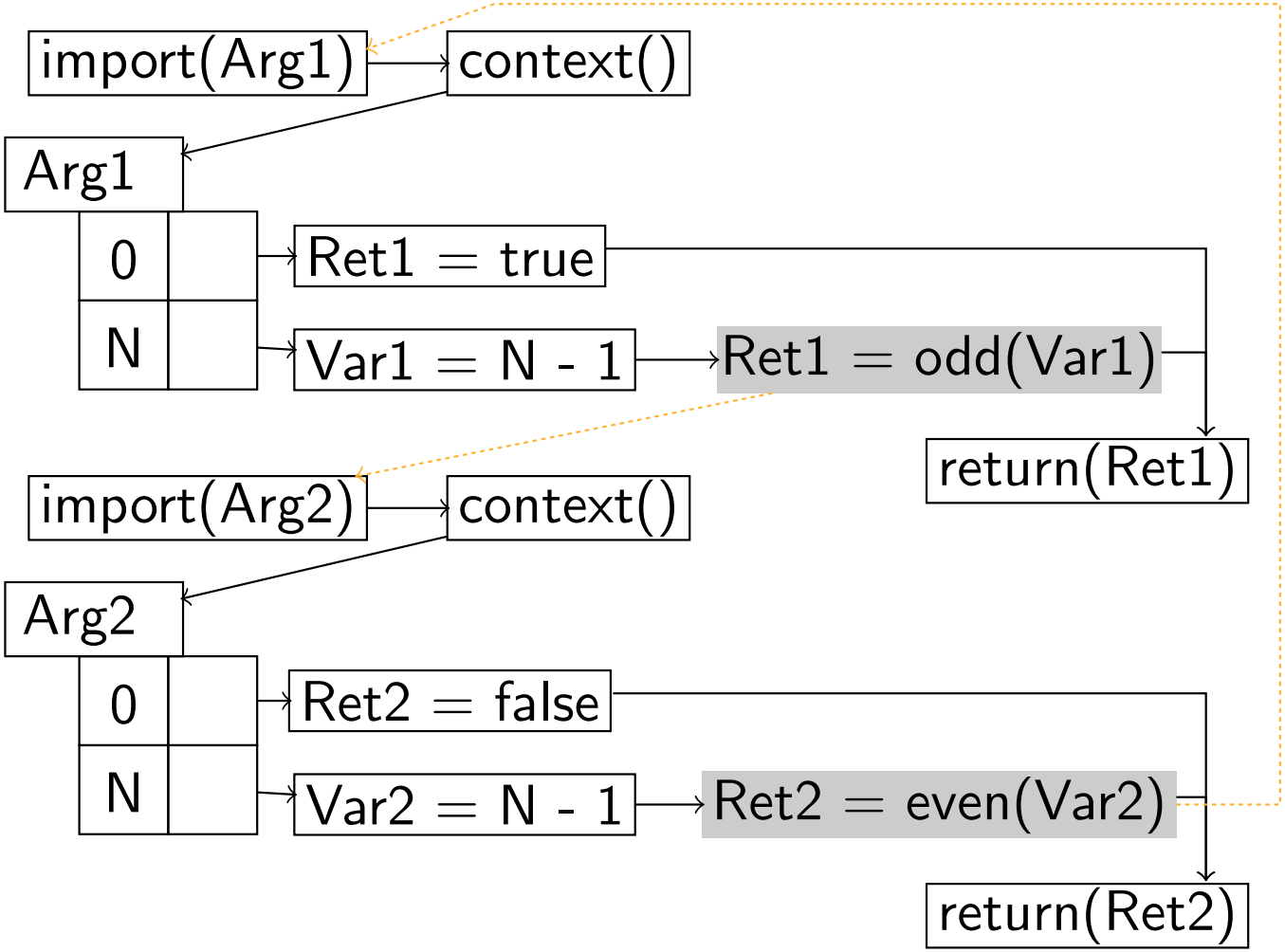
Import and context nodes are added to each function.

Definition and Generation of Flow Graphs



A return node is added to each function and the return value is bound to the return variable.

Definition and Generation of Flow Graphs



Call edges are inserted.

Call edges are bidirectional edges denoting call *and* return.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.

$\Rightarrow$ Data flow analysis is necessary.

Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.
$\Rightarrow$ Data flow analysis is necessary.
Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.
$\Rightarrow$ Data flow analysis is necessary.
Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.
$\Rightarrow$ Data flow analysis is necessary.
Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.
$\Rightarrow$ Data flow analysis is necessary.
Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.
$\Rightarrow$ Data flow analysis is necessary.
Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data Flow Analysis

The control flow can depend on the data flow in functional programming.
$\Rightarrow$ Data flow analysis is necessary.
Several ways of data flow must be handled during the analysis.

- Aliasing by copy expressions.
- Values from the context of a function which are fixed during fun generation.
- Values that are stored into and selected from a structure.
- Values that are sent to/received from a different process.

Data flow analysis considering these forms of data flow is powerful enough to compute accurate flow graphs.

Edge Types in Flow Graphs

Some edges in flow graphs depend on the data flow +, other edges do not −.

- − Neighborhood edges express neighborhood relations of expressions within a function.
- Call edges connect a call node and the import node of the called function.
  - − First order call edges emerging from from first order calls.
  - + Higher order call edges emerging from from higher order calls.
- + Throw edges connect a throw node with a corresponding catch node.
- + Message edges connect a send node with the corresponding receive node.

Edge Types in Flow Graphs

Some edges in flow graphs depend on the data flow +, other edges do not −.

- − Neighborhood edges express neighborhood relations of expressions within a function.
- Call edges connect a call node and the import node of the called function.
  - − First order call edges emerging from from first order calls.
  - + Higher order call edges emerging from from higher order calls.
- + Throw edges connect a throw node with a corresponding catch node.
- + Message edges connect a send node with the corresponding receive node.

Edge Types in Flow Graphs

Some edges in flow graphs depend on the data flow +, other edges do not −.

- − Neighborhood edges express neighborhood relations of expressions within a function.
- ▶ Call edges connect a call node and the import node of the called function.
  - − First order call edges emerging from from first order calls.
  - + Higher order call edges emerging from from higher order calls.
- + Throw edges connect a throw node with a corresponding catch node.
- + Message edges connect a send node with the corresponding receive node.

Edge Types in Flow Graphs

Some edges in flow graphs depend on the data flow +, other edges do not −.

- − Neighborhood edges express neighborhood relations of expressions within a function.
- ▶ Call edges connect a call node and the import node of the called function.
  - − First order call edges emerging from from first order calls.
  - + Higher order call edges emerging from from higher order calls.
- + Throw edges connect a throw node with a corresponding catch node.
- + Message edges connect a send node with the corresponding receive node.

Edge Types in Flow Graphs

Some edges in flow graphs depend on the data flow +, other edges do not −.

- − Neighborhood edges express neighborhood relations of expressions within a function.
- Call edges connect a call node and the import node of the called function.
  - − First order call edges emerging from from first order calls.
  - + Higher order call edges emerging from from higher order calls.
- + Throw edges connect a throw node with a corresponding catch node.
- + Message edges connect a send node with the corresponding receive node.

Higher Order Call Edges

Perform for each call node c with function given by the variable v

1. Compute all functions f in the flow graph potentially called by c.
   1.1 Determine all definitions d reaching the use of v in c.
   1.2 Use partial evaluation to determine the functions f expressed by each d.

   This process must be approximate.
2. Introduce a call edge from c to the import node of f.

Higher Order Call Edges

Perform for each call node c with function given by the variable v

1. Compute all functions f in the flow graph potentially called by c.
   1.1 Determine all definitions d reaching the use of v in c.
   1.2 Use partial evaluation to determine the functions f expressed by each d.

   This process must be approximate.
2. Introduce a call edge from c to the import node of f.

Higher Order Call Edges

Perform for each call node c with function given by the variable v

1. Compute all functions f in the flow graph potentially called by c.
   1.1 Determine all definitions d reaching the use of v in c.
   1.2 Use partial evaluation to determine the functions f expressed by each d.

   This process must be approximate.
2. Introduce a call edge from c to the import node of f.

Throw Edges

Perform for each catch node c

1. Search all reachable throw nodes t.
   - t is reachable from c if a path not containing another catch node exists.
   - Only neighborhood edges and call edges allowed.
   - Search is possible by DFS.
   - Collecting local information on function level can speed up the search.
2. Introduce a throw edge from t to c.

Throw Edges

Perform for each catch node c

1. Search all reachable throw nodes t.
   - t is reachable from c if a path not containing another catch node exists.
   - Only neighborhood edges and call edges allowed.
   - Search is possible by DFS.
   - Collecting local information on function level can speed up the search.
2. Introduce a throw edge from t to c.

Throw Edges

Perform for each catch node c

1. Search all reachable throw nodes t.
   - t is reachable from c if a path not containing another catch node exists.
   - Only neighborhood edges and call edges allowed.
   - Search is possible by DFS.
   - Collecting local information on function level can speed up the search.
2. Introduce a throw edge from t to c.

Message Edges

For each send node s compute the outgoing message edges as follows.

1. Compute the definitions d reaching the message variable of s.
2. Compute the possible destination processes P.
   $\rightarrow$ Data flow analysis and partial evaluation on the destination variable.
3. Determine all receive nodes r in each P.
4. Use abstract pattern matching between d and the patterns of r.
5. Insert a message edge from s to r if one d might match a pattern of r.

Message Edges

For each send node s compute the outgoing message edges as follows.

1. Compute the definitions d reaching the message variable of s.
2. Compute the possible destination processes P.
   $\rightarrow$ Data flow analysis and partial evaluation on the destination variable.
3. Determine all receive nodes r in each P.
4. Use abstract pattern matching between d and the patterns of r.
5. Insert a message edge from s to r if one d might match a pattern of r.

Message Edges

For each send node s compute the outgoing message edges as follows.

1. Compute the definitions d reaching the message variable of s.
2. Compute the possible destination processes P.
   $\rightarrow$ Data flow analysis and partial evaluation on the destination variable.
3. Determine all receive nodes r in each P.
4. Use abstract pattern matching between d and the patterns of r.
5. Insert a message edge from s to r if one d might match a pattern of r.

Message Edges

For each send node s compute the outgoing message edges as follows.

1. Compute the definitions d reaching the message variable of s.
2. Compute the possible destination processes P.
   $\rightarrow$ Data flow analysis and partial evaluation on the destination variable.
3. Determine all receive nodes r in each P.
4. Use abstract pattern matching between d and the patterns of r.
5. Insert a message edge from s to r if one d might match a pattern of r.

Message Edges

For each send node s compute the outgoing message edges as follows.

1. Compute the definitions d reaching the message variable of s.
2. Compute the possible destination processes P.
   $\rightarrow$ Data flow analysis and partial evaluation on the destination variable.
3. Determine all receive nodes r in each P.
4. Use abstract pattern matching between d and the patterns of r.
5. Insert a message edge from s to r if one d might match a pattern of r.

Message Edges

For each send node s compute the outgoing message edges as follows.

1. Compute the definitions d reaching the message variable of s.
2. Compute the possible destination processes P.
   $\rightarrow$ Data flow analysis and partial evaluation on the destination variable.
3. Determine all receive nodes r in each P.
4. Use abstract pattern matching between d and the patterns of r.
5. Insert a message edge from s to r if one d might match a pattern of r.

Dependences During Edge Computation

Data flow dependent edges depend on each other.

- Call edges depend on all types of edges.
- Throw edges depend on neighborhood edges and call edges.
- Message edges depend on all types of edges.

$\Rightarrow$ No order of computation stages that makes all necessary edge types available for each stage.

Dependences During Edge Computation

Data flow dependent edges depend on each other.

- Call edges depend on all types of edges.
- Throw edges depend on neighborhood edges and call edges.
- Message edges depend on all types of edges.

$\Rightarrow$ No order of computation stages that makes all necessary edge types available for each stage.

Dependences During Edge Computation

Data flow dependent edges depend on each other.

- Call edges depend on all types of edges.
- Throw edges depend on neighborhood edges and call edges.
- Message edges depend on all types of edges.

⇒ No order of computation stages that makes all necessary edge types available for each stage.

Dependences During Edge Computation

Data flow dependent edges depend on each other.

- Call edges depend on all types of edges.
- Throw edges depend on neighborhood edges and call edges.
- Message edges depend on all types of edges.

$\Rightarrow$ No order of computation stages that makes all necessary edge types available for each stage.

Dependences During Edge Computation

Data flow dependent edges depend on each other.

- Call edges depend on all types of edges.
- Throw edges depend on neighborhood edges and call edges.
- Message edges depend on all types of edges.

$\Rightarrow$ No order of computation stages that makes all necessary edge types available for each stage.

Iterated Edge Computation

- Three individual stages for computing the edge types are implemented.
- Iteration over all stages.
- A fixed point is reached when no set of edges is extended during one iteration.

Iterated Edge Computation

- Three individual stages for computing the edge types are implemented.
- Iteration over all stages.
- A fixed point is reached when no set of edges is extended during one iteration.

Iterated Edge Computation

- Three individual stages for computing the edge types are implemented.
- Iteration over all stages.
- A fixed point is reached when no set of edges is extended during one iteration.

Coverage Criteria Principle

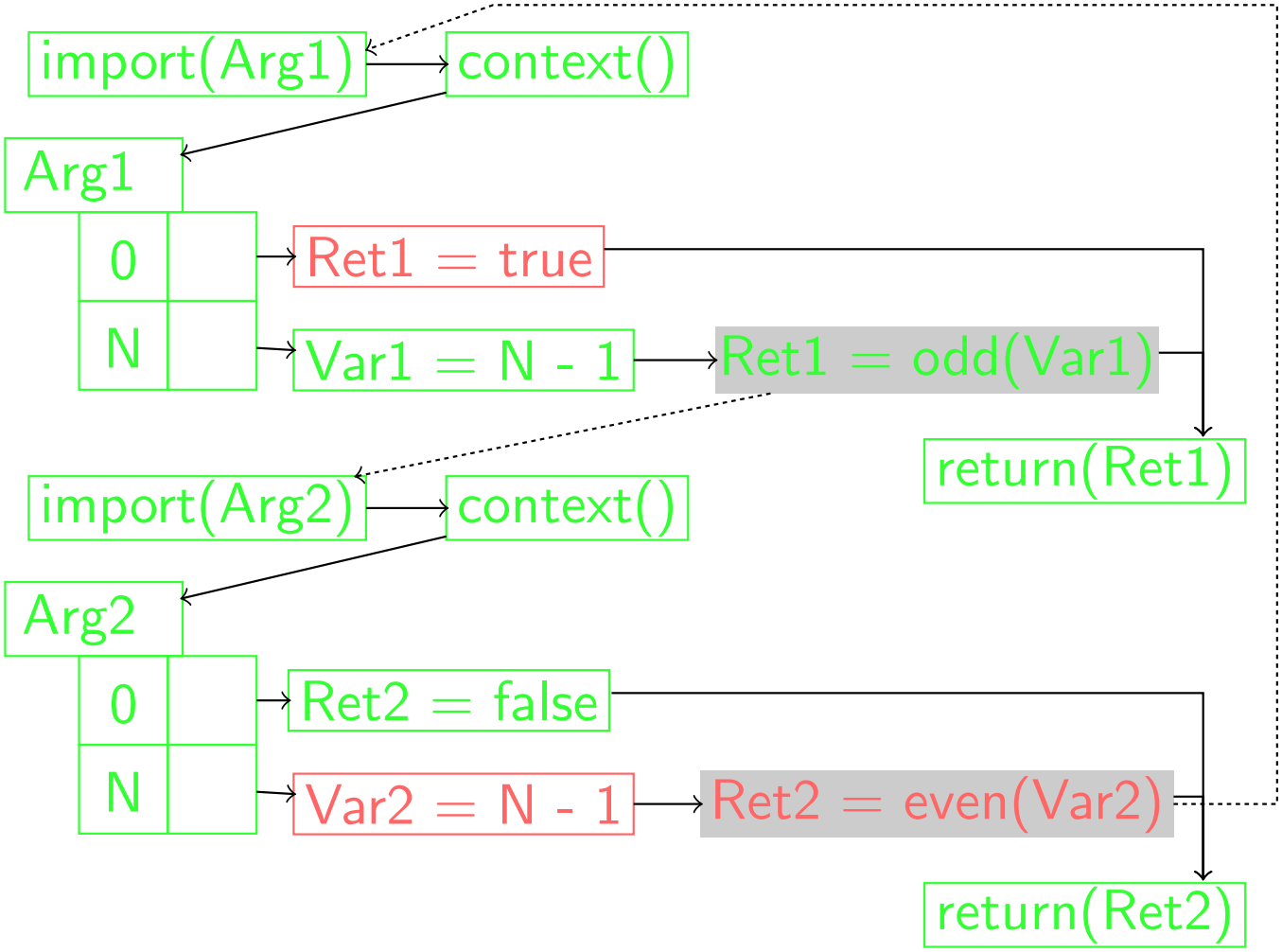
- Tests must be performed in a supervised manner.
- The resulting trace is analyzed according to some coverage criterion.

Coverage Criteria Principle

- Tests must be performed in a supervised manner.
- The resulting trace is analyzed according to some coverage criterion.

1. Node coverage criterion

Coverage Criteria: even(1)

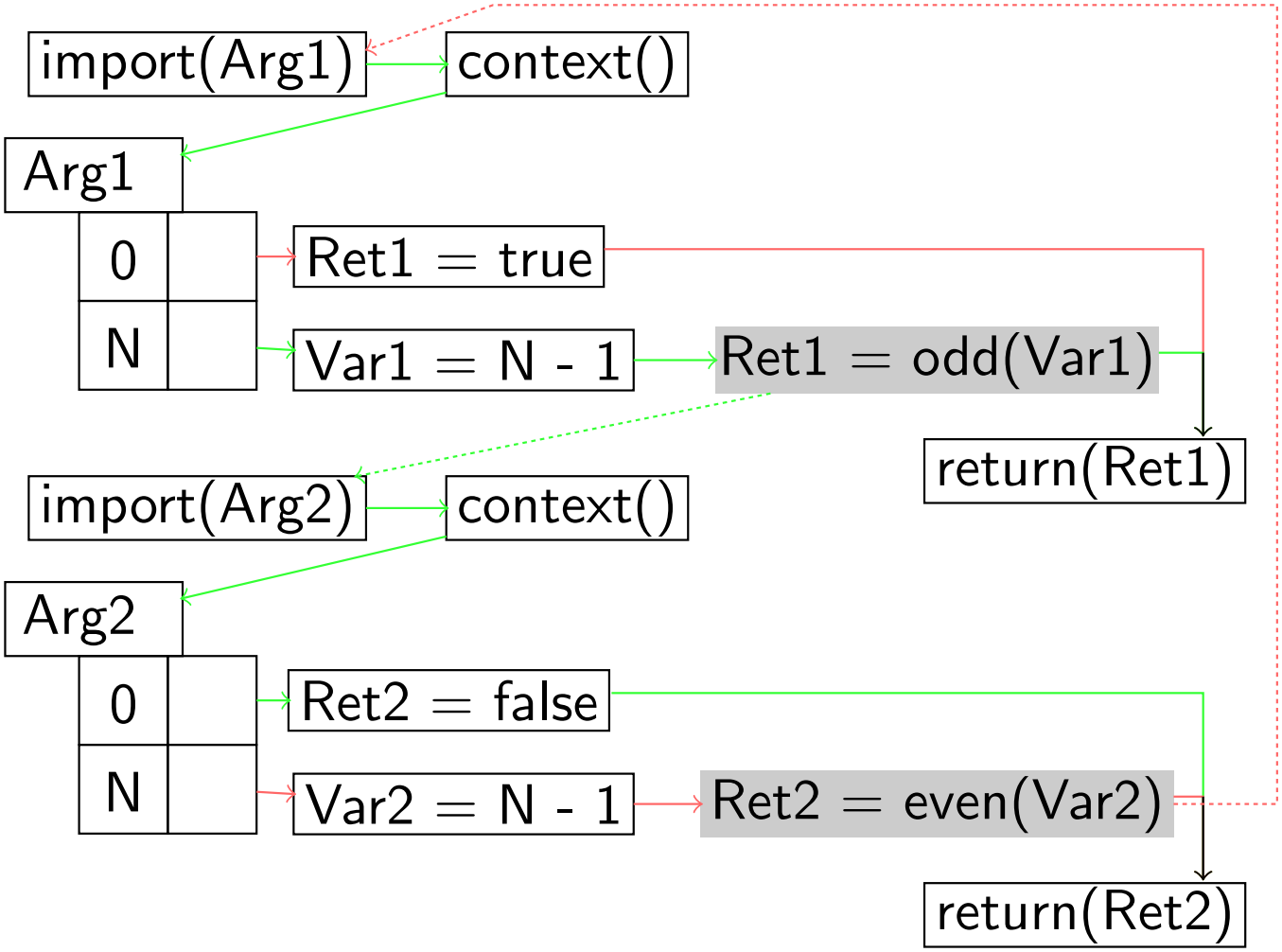


Coverage Criteria Principle

- Tests must be performed in a supervised manner.
- The resulting trace is analyzed according to some coverage criterion.

1. Node coverage criterion
2. Edge coverage criterion

Coverage Criteria: even(1)



Coverage Criteria Principle

- Tests must be performed in a supervised manner.
- The resulting trace is analyzed according to some coverage criterion.

1. Node coverage criterion
2. Edge coverage criterion
3. Data flow oriented coverage criteria seem to be best suited for functional programming.

Conclusion

- The presented flow graphs reflect the structure of the source code as accurately as possible.
- Special *call edges* represent the control flow during the call and during the return from the call.
- Data flow analysis is used to calculate call edges, throw edges, and message edges.
- Flow graphs can be computed for almost the whole Erlang standard OTP R9C by a prototype.

Conclusion

- The presented flow graphs reflect the structure of the source code as accurately as possible.
- Special *call edges* represent the control flow during the call and during the return from the call.
- Data flow analysis is used to calculate call edges, throw edges, and message edges.
- Flow graphs can be computed for almost the whole Erlang standard OTP R9C by a prototype.

Conclusion

- The presented flow graphs reflect the structure of the source code as accurately as possible.
- Special *call edges* represent the control flow during the call and during the return from the call.
- Data flow analysis is used to calculate call edges, throw edges, and message edges.
- Flow graphs can be computed for almost the whole Erlang standard OTP R9C by a prototype.

Conclusion

- The presented flow graphs reflect the structure of the source code as accurately as possible.
- Special *call edges* represent the control flow during the call and during the return from the call.
- Data flow analysis is used to calculate call edges, throw edges, and message edges.
- Flow graphs can be computed for almost the whole Erlang standard OTP R9C by a prototype.

Future Work

- Extension of the prototype implementation to the full Erlang standard.
  (List comprehensions missing up to now)
- Implementation of missing GUI elements.
  - Current drawing routine for flow graphs just handles neighborhood edges.
  - Useful arrangement of functions is an open problem.
- Extension of the system to OTP R10B.
- Evaluation of flow graph directed testing in a larger case study.

Future Work

- Extension of the prototype implementation to the full Erlang standard.
  (List comprehensions missing up to now)
- Implementation of missing GUI elements.
  - Current drawing routine for flow graphs just handles neighborhood edges.
  - Useful arrangement of functions is an open problem.
- Extension of the system to OTP R10B.
- Evaluation of flow graph directed testing in a larger case study.

Future Work

- Extension of the prototype implementation to the full Erlang standard.
  (List comprehensions missing up to now)
- Implementation of missing GUI elements.
  - Current drawing routine for flow graphs just handles neighborhood edges.
  - Useful arrangement of functions is an open problem.
- Extension of the system to OTP R10B.
- Evaluation of flow graph directed testing in a larger case study.

Future Work

- Extension of the prototype implementation to the full Erlang standard.
  (List comprehensions missing up to now)
- Implementation of missing GUI elements.
  - Current drawing routine for flow graphs just handles neighborhood edges.
  - Useful arrangement of functions is an open problem.
- Extension of the system to OTP R10B.
- Evaluation of flow graph directed testing in a larger case study.

Thank you.

Properties of the prototype.

- Covered language standard: almost OTP R9C.
  - List comprehensions are missing.
  - Includes of .hrl files not yet considered.
- No try constructs from OTP R10B.
- Graphical representation is incomplete.

Properties of the prototype.

- Covered language standard: almost OTP R9C.
  - List comprehensions are missing.
  - Includes of .hrl files not yet considered.
- No try constructs from OTP R10B.
- Graphical representation is incomplete.

Properties of the prototype.

- Covered language standard: almost OTP R9C.
  - List comprehensions are missing.
  - Includes of .hrl files not yet considered.
- No try constructs from OTP R10B.
- Graphical representation is incomplete.