

Lemmas on Demand for Lambdas

Mathias Preiner, Aina Niemetz and Armin Biere

Institute for Formal Models and Verification (FMV)
Johannes Kepler University, Linz, Austria
<http://fmv.jku.at/>

DIFTS Workshop 2013
October 19, 2013
Portland, OR, USA

Introduction

Why Lambdas?

Theory of arrays [McCarthy, 1962]

A1 $i = j \rightarrow \text{read}(a, i) = \text{read}(a, j)$ (array congruence)

A2 $i = j \rightarrow \text{read}(\text{write}(a, i, e), j) = e$ (read-over-write 1)

A3 $i \neq j \rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j)$ (read-over-write 2)

Limitations

- array operations restricted to single array indices
- no efficient modeling of parallel array updates (e.g.: *memset*, *memcpy*)

→ [Bryant et al., 2002] tackle limitations by using restricted λ -terms in UCLID

Introduction

Why Lambdas?

Theory of arrays [McCarthy, 1962]

A1 $i = j \rightarrow \text{read}(a, i) = \text{read}(a, j)$ (array congruence)

A2 $i = j \rightarrow \text{read}(\text{write}(a, i, e), j) = e$ (read-over-write 1)

A3 $i \neq j \rightarrow \text{read}(\text{write}(a, i, e), j) = \text{read}(a, j)$ (read-over-write 2)

Limitations

- array operations restricted to single array indices
- no efficient modeling of parallel array updates (e.g.: *memset*, *memcpy*)

→ [Bryant et al., 2002] tackle limitations by using restricted λ -terms in UCLID

Lambdas as arrays

- *write(a, i, e)*:
 $\lambda j . \text{ite}(i = j, e, \text{read}(a, j))$
- *memset(a, i, n, e)*:
 $\lambda j . \text{ite}(i \leq j \wedge j < i + n, e, \text{read}(a, j))$
- *memcpy(a, b, i, k, n)*:
 $\lambda j . \text{ite}(k \leq j \wedge j < k + n, \text{read}(a, i + j - k), \text{read}(b, j))$
- ...

Further applications

- ordered data structures
- arbitrary functions
- SMT-LIB v2 macros
- ...

Lambdas as arrays

- *write(a, i, e)*:
 $\lambda j . \text{ite}(i = j, e, \text{read}(a, j))$
 - *memset(a, i, n, e)*:
 $\lambda j . \text{ite}(i \leq j \wedge j < i + n, e, \text{read}(a, j))$
 - *memcpy(a, b, i, k, n)*:
 $\lambda j . \text{ite}(k \leq j \wedge j < k + n, \text{read}(a, i + j - k), \text{read}(b, j))$
 - ...
- can be symbolic

Further applications

- ordered data structures
- arbitrary functions
- SMT-LIB v2 macros
- ...

Lambdas as arrays

- *write(a, i, e)*:
 $\lambda j . \text{ite}(i = j, e, \text{read}(a, j))$
- *memset(a, i, n, e)*:
 $\lambda j . \text{ite}(i \leq j \wedge j < i + n, e, \text{read}(a, j))$
- *memcpy(a, b, i, k, n)*:
 $\lambda j . \text{ite}(k \leq j \wedge j < k + n, \text{read}(a, i + j - k), \text{read}(b, j))$
- ...

Further applications

- ordered data structures
- arbitrary functions
- SMT-LIB v2 macros
- ...

UCLID [[Seshia, 2005](#)]

- SMT solver using eager approach
- non-recursive λ -terms
- λ -terms used for modeling arrays and array operations (and more)

Lazy SMT solvers with lambda support:

- CVC4 [[Barrett et al., 2011](#)]
- Yices [[Dutertre and de Moura, 2006](#)]

Lambda handling in SMT solvers

- λ -terms treated as C-style macros
- eager elimination with β -reduction
- may result in an **exponential blow-up** in formula size

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$

$P(L_1(a))$



L_1



L_2



⋮



L_{k-1}



L_k

2^k instantiations of L_k

→ avoid with lazy lambda handling

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$

$P(L_1(a))$



L_1



L_2



⋮



L_{k-1}



L_k

2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$

$$P(L_1(a))$$



$$L_1$$

2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

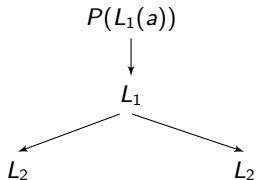
$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$



2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

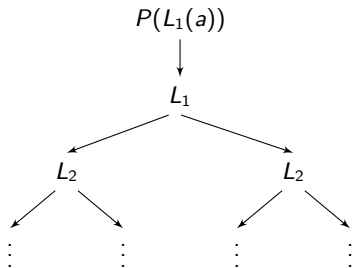
$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$



2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

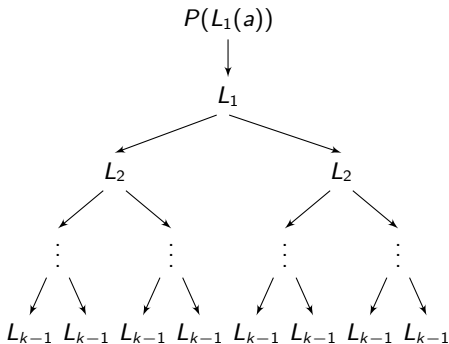
$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$



2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

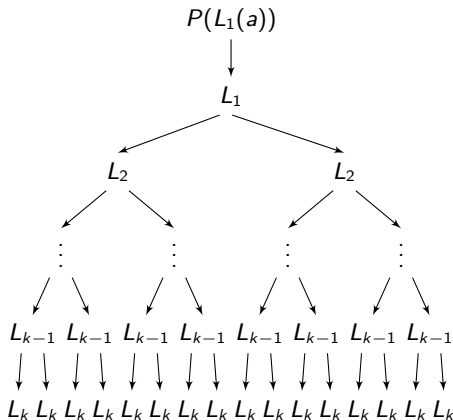
$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$



2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Introduction

Eager Lambda Elimination Worst-Case

Example [Seshia, 2005]

$$F := P(L_1(a))$$

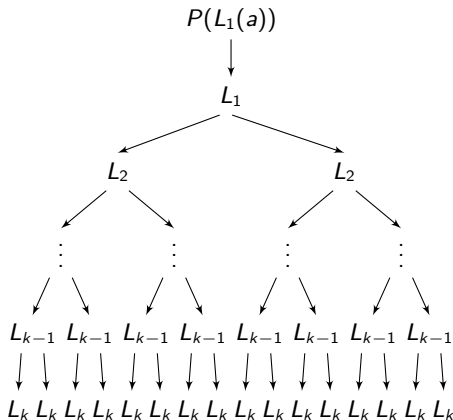
$$L_1 := \lambda x . f_1(L_2(x), L_2(g_1(x)))$$

$$L_2 := \lambda x . f_2(L_3(x), L_3(g_2(x)))$$

⋮

$$L_{k-1} := \lambda x . f_{k-1}(L_k(x), L_k(g_{k-1}(x)))$$

$$L_k := \lambda x . g_k(x)$$



2^k instantiations of L_k

→ avoid with **lazy lambda handling**

Boolector

- lazy SMT solver
- employs **lemmas on demand**
- supported theories:
 - fixed size bit vectors
 - arrays
- no quantifiers

Old version (pre-lambda)

- extensionality on arrays

New version

- λ -term support
- extensionality not supported (yet)

Extensionality on arrays

$$\begin{aligned} a &= b \\ \Leftrightarrow \\ \forall i . \text{read}(a, i) &= \text{read}(b, i) \end{aligned}$$

Extensionality on lambdas

$$\begin{aligned} \lambda \bar{x} . \phi &= \lambda \bar{y} . \psi \\ \Leftrightarrow \\ \forall \bar{a} . (\lambda \bar{x} . \phi)(\bar{a}) &= (\lambda \bar{y} . \psi)(\bar{a}) \end{aligned}$$

Quantifiers with extensionality on lambdas

$$\begin{aligned} \lambda x . p(x) &= \lambda x . \top \\ \Leftrightarrow \\ \forall x . p(x) \end{aligned}$$

Extensionality on arrays

$$\begin{aligned} a &= b \\ \Leftrightarrow \\ \forall i . \text{read}(a, i) &= \text{read}(b, i) \end{aligned}$$

Extensionality on lambdas

$$\begin{aligned} \lambda \bar{x} . \phi &= \lambda \bar{y} . \psi \\ \Leftrightarrow \\ \forall \bar{a} . (\lambda \bar{x} . \phi)(\bar{a}) &= (\lambda \bar{y} . \psi)(\bar{a}) \end{aligned}$$

Quantifiers with extensionality on lambdas

$$\begin{aligned} \lambda x . p(x) &= \lambda x . \top \\ \Leftrightarrow \\ \forall x . p(x) \end{aligned}$$

Extensionality on arrays

$$\begin{aligned}
 a &= b \\
 &\Leftrightarrow \\
 \forall i . \text{read}(a, i) &= \text{read}(b, i)
 \end{aligned}$$

Extensionality on lambdas

$$\begin{aligned}
 \lambda \bar{x} . \phi &= \lambda \bar{y} . \psi \\
 &\Leftrightarrow \\
 \forall \bar{a} . (\lambda \bar{x} . \phi)(\bar{a}) &= (\lambda \bar{y} . \psi)(\bar{a})
 \end{aligned}$$

Quantifiers with extensionality on lambdas

$$\begin{aligned}
 \lambda x . p(x) &= \lambda x . \top \\
 &\Leftrightarrow \\
 \forall x . p(x)
 \end{aligned}$$

Restrictions

- non-recursive
- non-extensional
- non-higher order functions

Lambdas in Boolector

- arrays represented as λ -terms \uninterpreted functions
 - no terms of sort array
 - uniform handling of arrays and λ -terms
- SMT-LIB v2 macros treated as curried λ -terms
- **lazy instantiation** of λ -terms
 - optional eager elimination
- new decision procedure DP_λ for λ -terms
 - generalization of array decision procedure [Brummayer and Biere, 2009]

Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

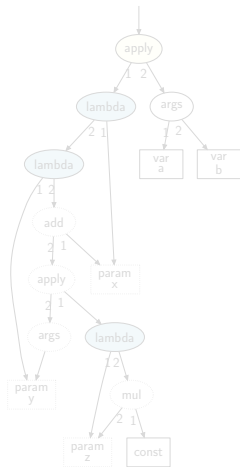
Full β -reduction

- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$



Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

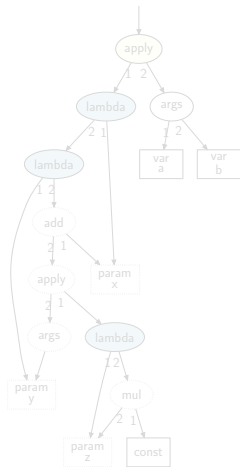
Full β -reduction

- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$



Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

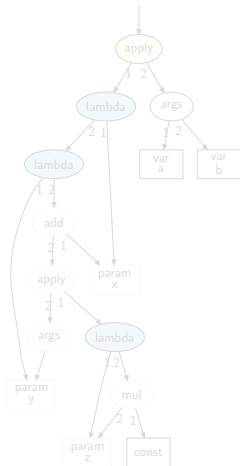
Full β -reduction

- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$



Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

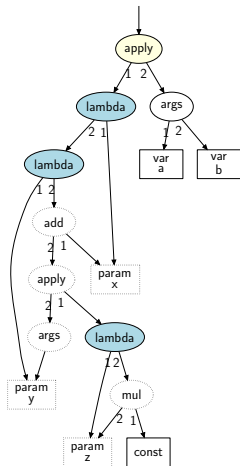
Full β -reduction

- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$



Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

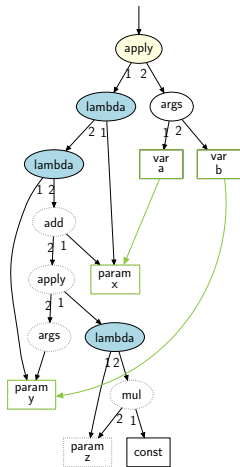
Full β -reduction

- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$



Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

Full β -reduction

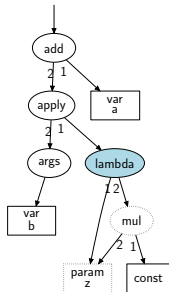
- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$

Result partial β -reduction



Lambdas in Boolector

β -reduction Approaches

Partial β -reduction

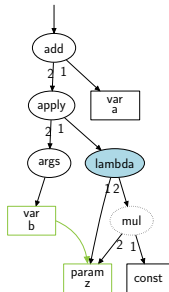
- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

Full β -reduction

- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters



Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$

Partial β -reduction

- like β -reduction in λ -calculus
- λ -terms are expanded "function-wise"
- required for consistency checking in DP_λ
→ considers current assignment

Full β -reduction

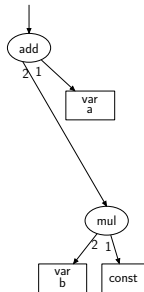
- eager elimination of λ -terms
- optional rewriting step

Given a DAG representing a λ -term ...

- 1 perform DFS post-order traversal
- 2 consecutively assign arguments to parameters
- 3 rebuild terms with arguments instead of parameters

Our notation for partial β -reduction: $\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$

Result full β -reduction



Refinement loop

- 1 abstract input formula ϕ (bit vector skeleton)
 - introduce **fresh** bit vector variable for each function application
 - translate bit vector skeleton into prop. formula
- 2 let SAT solver "guess" a solution
 - if SAT solver returns unsatisfiable, terminate with **unsatisfiable**
- 3 check if satisfying assignment is **consistent w.r.t.** ϕ ($consistent_\lambda$)
 - if check succeeds, terminate with **satisfiable**
- 4 if check fails, add lemma to **refine** formula abstraction ($lemma_\lambda$)
- 5 continue with 2

Lemmas on Demand for Lambdas

Consistency Checking in DP_λ

What to check?

Check whether current assignment σ is spurious or not

Rules

- rule C: function congruence axiom EUF

$$\forall \bar{a}, \bar{b}. \bigwedge_{i=1}^n \sigma(a_i) = \sigma(b_i) \rightarrow \sigma(f(\bar{a})) = \sigma(f(\bar{b}))$$
$$\dots \rightarrow \sigma(v_{f(\bar{a})}) = \sigma(v_{f(\bar{b})})$$

- rule B: abstraction variable consistency

$$\sigma(v_{\lambda_{\bar{x}}(a)}) = \sigma(\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p)$$

→ Optimization: rule P (see paper for more details)

Lemmas on Demand for Lambdas

Consistency Checking in DP_λ

What to check?

Check whether current assignment σ is spurious or not

Rules

- rule C: function congruence axiom EUF

$$\forall \bar{a}, \bar{b}. \bigwedge_{i=1}^n \sigma(a_i) = \sigma(b_i) \rightarrow \sigma(f(\bar{a})) = \sigma(f(\bar{b}))$$
$$\dots \rightarrow \sigma(v_{f(\bar{a})}) = \sigma(v_{f(\bar{b})})$$

- rule B: abstraction variable consistency

$$\sigma(v_{\lambda_{\bar{x}}(a)}) = \sigma(\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p)$$

→ Optimization: rule P (see paper for more details)

Lemmas on Demand for Lambdas

Consistency Checking in DP_λ

What to check?

Check whether current assignment σ is spurious or not

Rules

- rule C: function congruence axiom EUF

$$\forall \bar{a}, \bar{b}. \bigwedge_{i=1}^n \sigma(a_i) = \sigma(b_i) \rightarrow \sigma(f(\bar{a})) = \sigma(f(\bar{b}))$$
$$\dots \rightarrow \sigma(v_{f(\bar{a})}) = \sigma(v_{f(\bar{b})})$$

- rule B: abstraction variable consistency

$$\sigma(v_{\lambda_{\bar{x}}(a)}) = \sigma(\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p)$$

→ Optimization: rule P (see paper for more details)

Lemmas on Demand for Lambdas

Consistency Checking in DP_λ

What to check?

Check whether current assignment σ is spurious or not

Rules

- rule C: function congruence axiom EUF

$$\forall \bar{a}, \bar{b}. \bigwedge_{i=1}^n \sigma(a_i) = \sigma(b_i) \rightarrow \sigma(f(\bar{a})) = \sigma(f(\bar{b}))$$
$$\dots \rightarrow \sigma(v_{f(\bar{a})}) = \sigma(v_{f(\bar{b})})$$

- rule B: abstraction variable consistency

$$\sigma(v_{\lambda_{\bar{x}}(a)}) = \sigma(\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p)$$

→ Optimization: rule P (see paper for more details)

Lemmas on Demand for Lambdas

Consistency Checking in DP_λ

What to check?

Check whether current assignment σ is spurious or not

Rules

- rule C: function congruence axiom EUF

$$\forall \bar{a}, \bar{b}. \bigwedge_{i=1}^n \sigma(a_i) = \sigma(b_i) \rightarrow \sigma(f(\bar{a})) = \sigma(f(\bar{b}))$$
$$\dots \rightarrow \sigma(v_{f(\bar{a})}) = \sigma(v_{f(\bar{b})})$$

- rule B: abstraction variable consistency

$$\sigma(v_{\lambda_{\bar{x}}(a)}) = \sigma(\lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p)$$

→ Optimization: rule P (see paper for more details)

Lemmas on Demand for Lambdas

Consistency Checking in DP_λ (cont.)

Algorithm $consistent_\lambda$

- adaption of propagation algorithm in [Brummayer and Biere, 2009]
- associate each function application with resp. function
→ maintain hash table ρ for every function
- for each **pair** of function applications in ρ check rule C
- for each function application in ρ check rule B (λ -terms only)
- if a **conflict** occurs, generate a lemma ($lemma_\lambda$)
- otherwise, current assignment σ is **valid**

Lemmas on Demand for Lambdas

Lemma Generation

Violation of rule C

$s := g(a_1, \dots, a_n)$, $t := h(b_1, \dots, b_n) \in \rho(f)$ violate rule C

- 1 find propagation path p^s (p^t) from s (t) to f
- 2 collect all *ite* conditions c_0^s, \dots, c_j^s (c_0^t, \dots, c_j^t) on path p^s (p^t) that were \top under given assignment σ
- 3 collect all *ite* conditions c_0^s, \dots, c_k^s (c_0^t, \dots, c_m^t) on path p^s (p^t) that were \perp under given assignment σ

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \wedge \bigwedge_{i=0}^n a_i = b_i \rightarrow s = t$$

Lemmas on Demand for Lambdas

Lemma Generation

Violation of rule C

$s := g(a_1, \dots, a_n)$, $t := h(b_1, \dots, b_n) \in \rho(f)$ violate rule C

- 1 find propagation path p^s (p^t) from s (t) to f
- 2 collect all *ite* conditions c_0^s, \dots, c_j^s (c_0^t, \dots, c_j^t) on path p^s (p^t) that were \top under given assignment σ
- 3 collect all *ite* conditions c_0^s, \dots, c_k^s (c_0^t, \dots, c_m^t) on path p^s (p^t) that were \perp under given assignment σ

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \wedge \bigwedge_{i=0}^n a_i = b_i \rightarrow s = t$$

Prop. conditions s

Lemmas on Demand for Lambdas

Lemma Generation

Violation of rule C

$s := g(a_1, \dots, a_n)$, $t := h(b_1, \dots, b_n) \in \rho(f)$ violate rule C

- 1 find propagation path p^s (p^t) from s (t) to f
- 2 collect all *ite* conditions c_0^s, \dots, c_j^s (c_0^t, \dots, c_j^t) on path p^s (p^t) that were \top under given assignment σ
- 3 collect all *ite* conditions c_0^s, \dots, c_k^s (c_0^t, \dots, c_m^t) on path p^s (p^t) that were \perp under given assignment σ

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \wedge \bigwedge_{i=0}^n a_i = b_i \rightarrow s = t$$

Prop. conditions t

Lemmas on Demand for Lambdas

Lemma Generation

Violation of rule C

$s := g(a_1, \dots, a_n)$, $t := h(b_1, \dots, b_n) \in \rho(f)$ violate rule C

- 1 find propagation path p^s (p^t) from s (t) to f
- 2 collect all *ite* conditions c_0^s, \dots, c_j^s (c_0^t, \dots, c_j^t) on path p^s (p^t) that were \top under given assignment σ
- 3 collect all *ite* conditions c_0^s, \dots, c_k^s (c_0^t, \dots, c_m^t) on path p^s (p^t) that were \perp under given assignment σ

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \wedge \bigwedge_{i=0}^n a_i = b_i \rightarrow s = t$$

function congruence

Lemmas on Demand for Lambdas

Lemma Generation (cont.)

Violation of rule B

$s := \lambda_{\bar{y}}(a_1, \dots, a_n) \in \rho(\lambda_{\bar{x}})$, $t := \lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_{\mathbf{p}}$ violates rule B

- 1 collect conditions $c_0^s, \dots, c_j^s, c_0^s, \dots, c_k^s$ as before
- 2 collect all *ite* conditions c_0^t, \dots, c_l^t that evaluated to \top under given assignment σ while obtaining t
- 3 collect all *ite* conditions c_0^t, \dots, c_m^t that evaluated to \perp under given assignment σ while obtaining t

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \rightarrow s = t$$

Lemmas on Demand for Lambdas

Lemma Generation (cont.)

Violation of rule B

$s := \lambda_{\bar{y}}(a_1, \dots, a_n) \in \rho(\lambda_{\bar{x}})$, $t := \lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_{\mathbf{p}}$ violates rule B

- 1 collect conditions $c_0^s, \dots, c_j^s, c_0^s, \dots, c_k^s$ as before
- 2 collect all *ite* conditions c_0^t, \dots, c_l^t that evaluated to \top under given assignment σ while obtaining t
- 3 collect all *ite* conditions c_0^t, \dots, c_m^t that evaluated to \perp under given assignment σ while obtaining t

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \rightarrow s = t$$

Prop. conditions s

Lemmas on Demand for Lambdas

Lemma Generation (cont.)

Violation of rule B

$s := \lambda_{\bar{y}}(a_1, \dots, a_n) \in \rho(\lambda_{\bar{x}})$, $t := \lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_p$ violates rule B

- 1 collect conditions $c_0^s, \dots, c_j^s, c_0^s, \dots, c_k^s$ as before
- 2 collect all *ite* conditions c_0^t, \dots, c_l^t that evaluated to \top under given assignment σ while obtaining t
- 3 collect all *ite* conditions c_0^t, \dots, c_m^t that evaluated to \perp under given assignment σ while obtaining t

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \rightarrow s = t$$

Eval. conditions t

Lemmas on Demand for Lambdas

Lemma Generation (cont.)

Violation of rule B

$s := \lambda_{\bar{y}}(a_1, \dots, a_n) \in \rho(\lambda_{\bar{x}})$, $t := \lambda_{\bar{x}}[x_1 \setminus a_1, \dots, x_n \setminus a_n]_{\mathbf{p}}$ violates rule B

- 1 collect conditions $c_0^s, \dots, c_j^s, c_0^s, \dots, c_k^s$ as before
- 2 collect all *ite* conditions c_0^t, \dots, c_l^t that evaluated to \top under given assignment σ while obtaining t
- 3 collect all *ite* conditions c_0^t, \dots, c_m^t that evaluated to \perp under given assignment σ while obtaining t

Lemma

$$\bigwedge_{i=0}^j c_i^s \wedge \bigwedge_{i=0}^k \neg c_i^s \wedge \bigwedge_{i=0}^l c_i^t \wedge \bigwedge_{i=0}^m \neg c_i^t \rightarrow s = t$$

abstr. variable consistency

3 benchmark categories

- **crafted**: benchmarks with SMT-LIB v2 macros
- **SMT'12**: all non-extensional QF_AUFBV benchmarks used in SMT competition 2012
- **application**: instantiation benchmarks ¹ [Falke et al., 2013] generated with LLBMC (with and without λ -terms as arrays)

SMT Solvers

- Boolector: with DP_λ
- Boolector_{nop}: with DP_λ , but without rule P
- Boolector _{β} : with eager λ -term elimination
- Boolector_{sc12}: version submitted to SMT competition 2012
- CVC4 1.2, MathSAT 5.2.6, SONOLAR 2013-05-15, STP 1673 (svn revision), Z3 4.3.1

Machine Setup: 2.83Ghz Intel Core 2 Quad, 8GB memory, Ubuntu 12.04.2

¹<http://llbmc.org/files/downloads/vstte-2013.tgz>

Experiments

Category: crafted

	Solver	Solved	TO	MO	Time [10 ³ s]	Space [GB]
macro blow-up	Boolector	100	0	0	24.2	9.4
	Boolector _{nop}	100	0	0	18.2	8.4
	Boolector _{β}	28	49	23	91.5	160.0
	CVC4	21	0	79	95.7	551.6
	MathSAT	51	2	47	64.6	395.0
	SONOLAR	26	74	0	90.2	1.7
	Z3	21	0	79	95.0	552.2
fisher-yates SAT	Boolector	7	10	1	14.0	7.5
	Boolector _{nop}	4	13	1	17.3	7.0
	Boolector _{β}	6	1	11	15.0	76.4
	CVC4	5	1	12	15.7	83.6
	MathSAT	6	10	2	14.7	17.3
	SONOLAR	3	14	1	18.1	6.9
	Z3	6	12	0	14.8	0.2
fisher-yates UNSAT	Boolector	5	13	1	17.4	7.1
	Boolector _{nop}	4	14	1	18.2	6.9
	Boolector _{β}	9	0	10	12.1	72.0
	CVC4	3	4	12	19.2	82.1
	MathSAT	6	11	2	15.9	14.7
	SONOLAR	3	15	1	19.2	6.8
	Z3	10	9	0	11.2	2.2

Limits: time: 1200s, memory: 7GB

Penalty: TO: +1200s, MO: +1200s, +7GB

Experiments

Category: crafted

	Solver	Solved	TO	MO	Time [10 ³ s]	Space [GB]
macro blow-up	Boolector	100	0	0	24.2	9.4
	Boolector _{nop}	100	0	0	18.2	8.4
	Boolector _{β}	28	49	23	91.5	160.0
	CVC4	21	0	79	95.7	551.6
	MathSAT	51	2	47	64.6	395.0
	SONOLAR	26	74	0	90.2	1.7
	Z3	21	0	79	95.0	552.2
fisher-yates SAT	Boolector	7	10	1	14.0	7.5
	Boolector _{nop}	4	13	1	17.3	7.0
	Boolector _{β}	6	1	11	15.0	76.4
	CVC4	5	1	12	15.7	83.6
	MathSAT	6	10	2	14.7	17.3
	SONOLAR	3	14	1	18.1	6.9
	Z3	6	12	0	14.8	0.2
fisher-yates UNSAT	Boolector	5	13	1	17.4	7.1
	Boolector _{nop}	4	14	1	18.2	6.9
	Boolector _{β}	9	0	10	12.1	72.0
	CVC4	3	4	12	19.2	82.1
	MathSAT	6	11	2	15.9	14.7
	SONOLAR	3	15	1	19.2	6.8
	Z3	10	9	0	11.2	2.2

Limits: time: 1200s, memory: 7GB

Penalty: TO: +1200s, MO: +1200s, +7GB

Experiments

Category: crafted

	Solver	Solved	TO	MO	Time [10 ³ s]	Space [GB]
macro blow-up	Boolector	100	0	0	24.2	9.4
	Boolector _{nop}	100	0	0	18.2	8.4
	Boolector _{β}	28	49	23	91.5	160.0
	CVC4	21	0	79	95.7	551.6
	MathSAT	51	2	47	64.6	395.0
	SONOLAR	26	74	0	90.2	1.7
	Z3	21	0	79	95.0	552.2
fisher-yates SAT	Boolector	7	10	1	14.0	7.5
	Boolector _{nop}	4	13	1	17.3	7.0
	Boolector _{β}	6	1	11	15.0	76.4
	CVC4	5	1	12	15.7	83.6
	MathSAT	6	10	2	14.7	17.3
	SONOLAR	3	14	1	18.1	6.9
	Z3	6	12	0	14.8	0.2
fisher-yates UNSAT	Boolector	5	13	1	17.4	7.1
	Boolector _{nop}	4	14	1	18.2	6.9
	Boolector _{β}	9	0	10	12.1	72.0
	CVC4	3	4	12	19.2	82.1
	MathSAT	6	11	2	15.9	14.7
	SONOLAR	3	15	1	19.2	6.8
	Z3	10	9	0	11.2	2.2

Limits: time: 1200s, memory: 7GB

Penalty: TO: +1200s, MO: +1200s, +7GB

Experiments

Category: crafted

	Solver	Solved	TO	MO	Time [10 ³ s]	Space [GB]
macro blow-up	Boolector	100	0	0	24.2	9.4
	Boolector _{nop}	100	0	0	18.2	8.4
	Boolector _β	28	49	23	91.5	160.0
	CVC4	21	0	79	95.7	551.6
	MathSAT	51	2	47	64.6	395.0
	SONOLAR	26	74	0	90.2	1.7
	Z3	21	0	79	95.0	552.2
fisher-yates SAT	Boolector	7	10	1	14.0	7.5
	Boolector _{nop}	4	13	1	17.3	7.0
	Boolector _β	6	1	11	15.0	76.4
	CVC4	5	1	12	15.7	83.6
	MathSAT	6	10	2	14.7	17.3
	SONOLAR	3	14	1	18.1	6.9
	Z3	6	12	0	14.8	0.2
fisher-yates UNSAT	Boolector	5	13	1	17.4	7.1
	Boolector _{nop}	4	14	1	18.2	6.9
	Boolector _β	9	0	10	12.1	72.0
	CVC4	3	4	12	19.2	82.1
	MathSAT	6	11	2	15.9	14.7
	SONOLAR	3	15	1	19.2	6.8
	Z3	10	9	0	11.2	2.2

Limits: time: 1200s, memory: 7GB

Penalty: TO: +1200s, MO: +1200s, +7GB

Experiments

Category: SMT'12

	Solver	Solved	TO	MO	Time [10 ³ s]	Space [GB]
SMT'12	Boolector	139	10	0	19.9	14.8
	Boolector _{nop}	134	15	0	26.3	14.5
	Boolector _{β}	137	11	1	21.5	22.7
	Boolector _{sc12}	140	9	0	15.9	10.3

- Boolector solves 5 instances that Boolector _{β} couldn't
- Boolector _{β} solves 3 instances that Boolector couldn't
- combined they solve 2 instances that Boolector_{sc12} couldn't

Limits: time: 1200s, memory: 7GB

Penalty: TO: +1200s, MO: +1200s, +7GB

Experiments

Category: application

	Solver	Solved	TO	MO	Time [s]	Space [MB]
instantiation	Boolector	37	8	0	576	235
	Boolector _{nop}	35	10	0	673	196
	Boolector _{β}	44	1	0	138	961
	Boolector _{sc12}	39	6	0	535	308
	STP	44	1	0	141	3814
lambda ²	Boolector	37	8	0	594	236
	Boolector _{nop}	35	10	0	709	166
	Boolector _{β}	45	0	0	52	676
	Boolector _{sc12}	-	-	-	-	-
	STP	-	-	-	-	-

Limits: time: 60s, memory: 7GB

Penalty: TO: +60s, MO: +60s, +7GB

²lambda benchmarks kindly provided by Carsten Sinz et. al.

Experiments

Category: application

	Solver	Solved	TO	MO	Time [s]	Space [MB]
instantiation	Boolector	37	8	0	576	235
	Boolector _{nop}	35	10	0	673	196
	Boolector _{β}	44	1	0	138	961
	Boolector _{sc12}	39	6	0	535	308
	STP	44	1	0	141	3814
lambda ²	Boolector	37	8	0	594	236
	Boolector _{nop}	35	10	0	709	166
	Boolector _{β}	45	0	0	52	676
	Boolector _{sc12}	-	-	-	-	-
	STP	-	-	-	-	-

Limits: time: 60s, memory: 7GB

Penalty: TO: +60s, MO: +60s, +7GB

²lambda benchmarks kindly provided by Carsten Sinz et. al.

Experiments

Category: application

	Solver	Solved	TO	MO	Time [s]	Space [MB]
instantiation	Boolector	37	8	0	576	235
	Boolector _{nop}	35	10	0	673	196
	Boolector _{β}	44	1	0	138	961
	Boolector _{sc12}	39	6	0	535	308
	STP	44	1	0	141	3814
lambda ²	Boolector	37	8	0	594	236
	Boolector _{nop}	35	10	0	709	166
	Boolector _{β}	45	0	0	52	676
	Boolector _{sc12}	-	-	-	-	-
	STP	-	-	-	-	-

Limits: time: 60s, memory: 7GB

Penalty: TO: +60s, MO: +60s, +7GB

²lambda benchmarks kindly provided by Carsten Sinz et. al.

Conclusion

- DP_λ , a decision procedure for non-recursive, non-extensional λ -terms
→ *consistent* $_\lambda$, *lemma* $_\lambda$
- experimental results look promising
→ category application demonstrates potential of native λ -term support
- still room for improvements
→ optimization of β -reduction
→ no λ -term specific rewriting yet

Future Work

- rewriting rules for λ -terms
- better β -reduction implementation
- various β -reduction strategies
- extensionality on λ -terms
- quantifiers







Lemmas on Demand for Lambdas

Mathias Preiner, Aina Niemetz and Armin Biere

Institute for Formal Models and Verification (FMV)
Johannes Kepler University, Linz, Austria
<http://fmv.jku.at/>

DIFTS Workshop 2013
October 19, 2013
Portland, OR, USA

References

-  Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., and Tinelli, C. (2011).
CVC4.
In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer.
-  Brummayer, R. and Biere, A. (2009).
Lemmas on Demand for the Extensional Theory of Arrays.
JSAT, 6(1-3):165–201.
-  Bryant, R. E., Lahiri, S. K., and Seshia, S. A. (2002).
Modeling and Verifying Systems Using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions.
In *CAV*, volume 2404 of *LNCS*, pages 78–92. Springer.
-  Dutertre, B. and de Moura, L. (2006).
The Yices SMT solver.
Tool paper at <http://yices.csl.sri.com/tool-paper.pdf>.
-  Falke, S., Merz, F., and Sinz, C. (2013).
Extending the Theory of Arrays: memset, memcpy, and Beyond.
In *Proc. VSTTE'13*.
-  McCarthy, J. (1962).
Towards a Mathematical Science of Computation. 