Efficiency refinements in the Dialectica interpretation

Trifon Trifonov

Faculty of Mathematics and Informatics, Sofia University
Supported by the European Social Fund within project BG051PO001-3.3.04/28.08.2009.

Computability in Europe 2011
Sofia, 29.06.2011

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\to$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\to$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\to$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Program extraction

From constructive proofs:

- Curry-Howard correspondence
- proofs as functional programs with verification
- modified realisability: proof $\rightarrow$ program $+$ certificate

Why would we want to extract from non-constructive proofs?

- sometimes they are easier than constructive ones
  - undecidable case distinctions
- sometimes we have no hope for an efficient algorithm
  - NP-complete problems
- sometimes provide more interesting solutions
  - use of continuations and accumulating parameters

Gödel's Dialectica interpretation

- Formulas A are problems
- Formulas specify the type of the solution $t : \tau^{+}(A)$
- Solutions are challenged by terms $y : \tau^{-}(A)$
- Translations specify when t is a solution of A for a challenge y ($|A|^{t}_{y}$)
- Proposed by Gödel (1958)
- Motivation: interpret classical arithmetic in a quantifier-free constructive system with higher types.

Gödel's Dialectica interpretation

- Formulas A are problems
- Formulas specify the type of the solution $t : \tau^+(A)$
- Solutions are challenged by terms $y : \tau^-(A)$
- Translations specify when t is a solution of A for a challenge y ($|A|^t_y$)
- Proposed by Gödel (1958)
- Motivation: interpret classical arithmetic in a quantifier-free constructive system with higher types.

Gödel's Dialectica interpretation

- Formulas A are problems
- Formulas specify the type of the solution $t : \tau^+(A)$
- Solutions are challenged by terms $y : \tau^-(A)$
- Translations specify when t is a solution of A for a challenge y ($|A|_y^t$)
- Proposed by Gödel (1958)
- Motivation: interpret classical arithmetic in a quantifier-free constructive system with higher types.

Gödel's Dialectica interpretation

- Formulas A are problems
- Formulas specify the type of the solution $t : \tau^{+}(A)$
- Solutions are challenged by terms $y : \tau^{-}(A)$
- Translations specify when t is a solution of A for a challenge y ($|A|_{y}^{t}$)
- Proposed by Gödel (1958)
- Motivation: interpret classical arithmetic in a quantifier-free constructive system with higher types.

Gödel's Dialectica interpretation

- Formulas A are problems
- Formulas specify the type of the solution $t : \tau^{+}(A)$
- Solutions are challenged by terms $y : \tau^{-}(A)$
- Translations specify when t is a solution of A for a challenge y ($|A|_y^t$)
- Proposed by Gödel (1958)
- Motivation: interpret classical arithmetic in a quantifier-free constructive system with higher types.

Gödel's Dialectica interpretation

- Formulas A are problems
- Formulas specify the type of the solution $t : \tau^+(A)$
- Solutions are challenged by terms $y : \tau^-(A)$
- Translations specify when t is a solution of A for a challenge y ($|A|^t_y$)
- Proposed by Gödel (1958)
- Motivation: interpret classical arithmetic in a quantifier-free constructive system with higher types.

Efficiency problems

1. every object term used in the proof is copied multiple times
   - terms appear in $\forall$ elimination
   - copied once for every occurrence of the quantified variable
   - reason: substitution is used on the meta-level
2. some computations are only used for verification
   - example: is the extracted function invertible?
   - to assert this we may need to compute its inverse
   - but the extracted program need not compute it
3. same conditions are checked multiple times
   - assumptions can be used more than once in a proof
   - Dialectica combines two counterexamples into one
   - if one of them is verified, no need to check further

Efficiency problems

1. every object term used in the proof is copied multiple times
   - terms appear in $\forall$ elimination
   - copied once for every occurrence of the quantified variable
   - reason: substitution is used on the meta-level
2. some computations are only used for verification
   - example: is the extracted function invertible?
   - to assert this we may need to compute its inverse
   - but the extracted program need not compute it
3. same conditions are checked multiple times
   - assumptions can be used more than once in a proof
   - Dialectica combines two counterexamples into one
   - if one of them is verified, no need to check further

Efficiency problems

1. every object term used in the proof is copied multiple times
   - terms appear in $\forall$ elimination
   - copied once for every occurrence of the quantified variable
   - reason: substitution is used on the meta-level
2. some computations are only used for verification
   - example: is the extracted function invertible?
   - to assert this we may need to compute its inverse
   - but the extracted program need not compute it
3. same conditions are checked multiple times
   - assumptions can be used more than once in a proof
   - Dialectica combines two counterexamples into one
   - if one of them is verified, no need to check further

Efficiency improvements

1. **let** definitions instead of substitutions
   - $(\lambda_x s)t$ instead of $s\,[x := t]$
   - a proof cut should be translated to an application
   - programmer's slang: **local variables**
2. annotate some parameters as computationally uniform
   - introduced by Berger, adapted by Hernest
   - automatic annotation (Ratiu & Schwichtenberg)
   - programmer's slang: **dead code cleanup**
3. remember counterexample checks
   - annotate every counterexample with a boolean flag
   - do not compute new counterexamples after one is found
   - programmers slang: **memoization**

Efficiency improvements

1. **let** definitions instead of substitutions
   - $(\lambda_x s)t$ instead of $s\,[x := t]$
   - a proof cut should be translated to an application
   - programmer's slang: **local variables**
2. annotate some parameters as computationally uniform
   - introduced by Berger, adapted by Hernest
   - automatic annotation (Ratiu & Schwichtenberg)
   - programmer's slang: **dead code cleanup**
3. remember counterexample checks
   - annotate every counterexample with a boolean flag
   - do not compute new counterexamples after one is found
   - programmers slang: **memoization**

Efficiency improvements

1. **let** definitions instead of substitutions
   - $(\lambda_x s)t$ instead of $s\,[x := t]$
   - a proof cut should be translated to an application
   - programmer's slang: **local variables**
2. annotate some parameters as computationally uniform
   - introduced by Berger, adapted by Hernest
   - automatic annotation (Ratiu & Schwichtenberg)
   - programmer's slang: **dead code cleanup**
3. remember counterexample checks
   - annotate every counterexample with a boolean flag
   - do not compute new counterexamples after one is found
   - programmers slang: **memoization**

Exponential example

Consider a proof of totality:

$$\forall_{x,y}\exists_z\big(z = 2^x(x+y)\big)$$

Extracted programs:

- Dialectica:

$$\begin{aligned} f(0,y) &= y \\ f(x+1,y) &= f(x,y+1) + f(x,y+1) \end{aligned}$$

- Modified realisability:

$$\begin{aligned} f(0,y) &= y \\ f(x+1,y) &= (\lambda_{z,g} g(z))(f(x,y+1))(\lambda_z z+z) \end{aligned}$$

Exponential example

Consider a proof of totality:

$$\forall_{x,y}\exists_z\big(z = 2^x(x+y)\big)$$

Extracted programs:

- Dialectica:

$$\begin{aligned} f(0,y) &= y \\ f(x+1,y) &= f(x,y+1) + f(x,y+1) \end{aligned}$$

- Modified realisability:

$$\begin{aligned} f(0,y) &= y \\ f(x+1,y) &= (\lambda_{z,g} g(z))(f(x,y+1))(\lambda_z z+z) \end{aligned}$$

Exponential example

Consider a proof of totality:

$$\forall_{x,y}\exists_z\big(z = 2^x(x+y)\big)$$

Extracted programs:

- Dialectica:

$$\begin{aligned} f(0,y) &= y \\ f(x+1,y) &= f(x,y+1) + f(x,y+1) \end{aligned}$$

- Modified realisability:

$$\begin{aligned} f(0,y) &= y \\ f(x+1,y) &= (\lambda_{z,g} g(z))(f(x,y+1))(\lambda_z z+z) \end{aligned}$$

Avoid repetition

Definition contexts

$$E \quad ::= \quad []^{\diamond} \mid (E^{\rho\Rightarrow\sigma} t^{\rho})^{\sigma} \mid (\lambda_{x^{\rho}} E^{\sigma})^{\rho\Rightarrow\sigma},$$
$$E[t^{\rho}] := E\,[\diamond := \rho]\,[[] := t]$$

- Let $\mathcal{P}$ be a proof of A from assumptions C_i
- Split extracted terms in two parts
- Binding term (context) $[\![\mathcal{P}]\!] : \sigma^{-}(A) \Rightarrow \diamond$
- Context-dependent terms $[\![\mathcal{P}]\!]^{+} : \sigma^{+}(A)$, $[\![\mathcal{P}]\!]_i^{-} : \sigma^{-}(C_i)$
- Combine at the end:

$$\{\![\mathcal{P}]\!\} := [\![\mathcal{P}]\!][\langle [\![\mathcal{P}]\!]^{+}, [\![\mathcal{P}]\!]_i^{-}, \dots \rangle]$$

Avoid repetition

Definition contexts

$$E \quad ::= \quad []^{\diamond} \mid (E^{\rho\Rightarrow\sigma} t^{\rho})^{\sigma} \mid (\lambda_{x^{\rho}} E^{\sigma})^{\rho\Rightarrow\sigma},$$
$$E[t^{\rho}] := E\,[\diamond := \rho]\,[[] := t]$$

- Let $\mathcal{P}$ be a proof of A from assumptions C_i
- Split extracted terms in two parts
- Binding term (context) $[\![\mathcal{P}]\!] : \sigma^{-}(A) \Rightarrow \diamond$
- Context-dependent terms $[\![\mathcal{P}]\!]^{+} : \sigma^{+}(A)$, $[\![\mathcal{P}]\!]_i^{-} : \sigma^{-}(C_i)$
- Combine at the end:

$$\{\!|\mathcal{P}|\!\} := [\![\mathcal{P}]\!][\langle [\![\mathcal{P}]\!]^{+}, [\![\mathcal{P}]\!]_i^{-}, \dots \rangle]$$

Avoid repetition

Definition contexts

$$E \quad ::= \quad []^{\diamond} \mid (E^{\rho\Rightarrow\sigma} t^{\rho})^{\sigma} \mid (\lambda_{x^{\rho}} E^{\sigma})^{\rho\Rightarrow\sigma},$$
$$E[t^{\rho}] := E\,[\diamond := \rho]\,[[] := t]$$

- Let $\mathcal{P}$ be a proof of A from assumptions C_i
- Split extracted terms in two parts
- Binding term (context) $[\![\mathcal{P}]\!] : \sigma^{-}(A) \Rightarrow \diamond$
- Context-dependent terms $[\![\mathcal{P}]\!]^{+} : \sigma^{+}(A)$, $[\![\mathcal{P}]\!]_i^{-} : \sigma^{-}(C_i)$
- Combine at the end:

$$\{\!|\mathcal{P}|\!\} := [\![\mathcal{P}]\!][\langle [\![\mathcal{P}]\!]^{+}, [\![\mathcal{P}]\!]_i^{-}, \ldots \rangle]$$

Avoid repetition

Definition contexts

$$
\begin{aligned}
E \quad &::= \quad []^{\diamond} \mid (E^{\rho \Rightarrow \sigma} t^{\rho})^{\sigma} \mid (\lambda_{x^{\rho}} E^{\sigma})^{\rho \Rightarrow \sigma}, \\
E[t^{\rho}] &:= E\,[\diamond := \rho]\,[[] := t]
\end{aligned}
$$

- Let $\mathcal{P}$ be a proof of A from assumptions C_i
- Split extracted terms in two parts
- Binding term (context) $[\![\mathcal{P}]\!] : \sigma^{-}(A) \Rightarrow \diamond$
- Context-dependent terms $[\![\mathcal{P}]\!]^{+} : \sigma^{+}(A)$, $[\![\mathcal{P}]\!]_i^{-} : \sigma^{-}(C_i)$
- Combine at the end:

$$
\{\!|\mathcal{P}|\!\} := [\![\mathcal{P}]\!][\langle [\![\mathcal{P}]\!]^{+}, [\![\mathcal{P}]\!]_i^{-}, \ldots \rangle]
$$

Avoid repetition

Definition contexts

$$
\begin{aligned}
E \quad &::= \quad []^{\diamond} \mid (E^{\rho\Rightarrow\sigma} t^{\rho})^{\sigma} \mid (\lambda_{x^{\rho}} E^{\sigma})^{\rho\Rightarrow\sigma}, \\
E[t^{\rho}] &:= E\,[\diamond := \rho]\,[[] := t]
\end{aligned}
$$

- Let $\mathcal{P}$ be a proof of A from assumptions C_i
- Split extracted terms in two parts
- Binding term (context) $[\![\mathcal{P}]\!] : \sigma^{-}(A) \Rightarrow \diamond$
- Context-dependent terms $[\![\mathcal{P}]\!]^{+} : \sigma^{+}(A)$, $[\![\mathcal{P}]\!]_i^{-} : \sigma^{-}(C_i)$
- Combine at the end:

$$\{\!|\mathcal{P}|\!\} := [\![\mathcal{P}]\!][\langle [\![\mathcal{P}]\!]^{+}, [\![\mathcal{P}]\!]_i^{-}, \ldots \rangle]$$

Avoid repetition

Definition contexts

$$
\begin{aligned}
E \quad &::= \quad []^{\diamond} \mid (E^{\rho\Rightarrow\sigma} t^{\rho})^{\sigma} \mid (\lambda_{x^{\rho}} E^{\sigma})^{\rho\Rightarrow\sigma}, \\
E[t^{\rho}] &:= E\,[\diamond := \rho]\,[[] := t]
\end{aligned}
$$

- Let $\mathcal{P}$ be a proof of A from assumptions C_i
- Split extracted terms in two parts
- Binding term (context) $[\![\mathcal{P}]\!] : \sigma^{-}(A) \Rightarrow \diamond$
- Context-dependent terms $[\![\mathcal{P}]\!]^{+} : \sigma^{+}(A)$, $[\![\mathcal{P}]\!]_i^{-} : \sigma^{-}(C_i)$
- Combine at the end:

$$\{\!|\mathcal{P}|\!\} := [\![\mathcal{P}]\!][\langle [\![\mathcal{P}]\!]^{+}, [\![\mathcal{P}]\!]_i^{-}, \ldots \rangle]$$

Quasi-linear extraction

In theory:

Theorem

Let $\mathcal{P}$ be a proof of A. Then the extracted program $\{\!|\mathcal{P}|\!\}^+$ is a witness of A and $\mathrm{size}(\{\!|\mathcal{P}|\!\}^+) = O(\mathrm{size}(\mathcal{P}) \cdot \mathrm{msl}(\mathcal{P})^2)$.

In practice:

$$
\begin{aligned}
& R := \lambda_{y_6} \textbf{let } x := y_6 \textbf{ in } \lambda_{y_7} \textbf{let } f_1 := s \textbf{ in } f_1 y_7, \text{ where} \\
& s := \mathcal{R}\, x\, t_0\, (\lambda_x \lambda_p \textbf{let } x_p := p \textbf{ in } t_1), \\
& t_0 := \lambda_{y_0} \textbf{let } y := y_0 \textbf{ in let } y_1 := y \textbf{ in } y_1, \\
& t_1 := \lambda_{y_2} \textbf{let } f_0 := (\lambda_{y_3} \textbf{let } z := y_3 \textbf{ in let } y_4 := z + z \textbf{ in } y_4) \textbf{ in} \\
& \qquad \textbf{let } y := y_2 \textbf{ in let } y_5 := y + 1 \textbf{ in let } z_0 := x_p y_5 \textbf{ in } f_0 z_0.
\end{aligned}
$$

Quasi-linear extraction

In theory:

Theorem

Let $\mathcal{P}$ be a proof of A. Then the extracted program $\{\!|\mathcal{P}|\!\}^+$ is a witness of A and $\mathrm{size}(\{\!|\mathcal{P}|\!\}^+) = O(\mathrm{size}(\mathcal{P}) \cdot \mathrm{msl}(\mathcal{P})^2)$.

In practice:

$$R := \lambda_{y_6} \textbf{let } x := y_6 \textbf{ in } \lambda_{y_7} \textbf{let } f_1 := s \textbf{ in } f_1 y_7, \text{ where}$$

$$s := \mathcal{R}\, x\, t_0\, (\lambda_x \lambda_p \textbf{let } x_p := p \textbf{ in } t_1),$$

$$t_0 := \lambda_{y_0} \textbf{let } y := y_0 \textbf{ in let } y_1 := y \textbf{ in } y_1,$$

$$t_1 := \lambda_{y_2} \textbf{let } f_0 := (\lambda_{y_3} \textbf{let } z := y_3 \textbf{ in let } y_4 := z + z \textbf{ in } y_4) \textbf{ in}$$

$$\textbf{let } y := y_2 \textbf{ in let } y_5 := y + 1 \textbf{ in let } z_0 := x_p y_5 \textbf{ in } f_0 z_0.$$

Quasi-linear extraction

In theory:

Theorem
Let $\mathcal{P}$ be a proof of A. Then the extracted program $\{\!|\mathcal{P}|\!\}^+$ is a witness of A and $\mathrm{size}(\{\!|\mathcal{P}|\!\}^+) = O(\mathrm{size}(\mathcal{P}) \cdot \mathrm{msl}(\mathcal{P})^2)$.

In practice:

$R := \lambda_{y_6} \textbf{let } x := y_6 \textbf{ in } \lambda_{y_7} \textbf{let } f_1 := s \textbf{ in } f_1 y_7,$ where

$s := \mathcal{R}\, x\, t_0\, (\lambda_x \lambda_p \textbf{let } x_p := p \textbf{ in } t_1),$

$t_0 := \lambda_{y_0} \textbf{let } y := y_0 \textbf{ in let } y_1 := y \textbf{ in } y_1,$

$t_1 := \lambda_{y_2} \textbf{let } f_0 := (\lambda_{y_3} \textbf{let } z := y_3 \textbf{ in let } y_4 := z + z \textbf{ in } y_4) \textbf{ in}$
$\textbf{let } y := y_2 \textbf{ in let } y_5 := y + 1 \textbf{ in let } z_0 := x_p y_5 \textbf{ in } f_0 z_0.$

Quasi-linear extraction

In theory:

Theorem

Let $\mathcal{P}$ be a proof of A. Then the extracted program $\{\!|\mathcal{P}|\!\}^+$ is a witness of A and $\mathrm{size}(\{\!|\mathcal{P}|\!\}^+) = O(\mathrm{size}(\mathcal{P}) \cdot \mathrm{msl}(\mathcal{P})^2)$*.*

In practice:

$$
\begin{aligned}
R &:= \lambda_{y_6} \textbf{let } x := y_6 \textbf{ in } \lambda_{y_7} \textbf{let } f_1 := s \textbf{ in } f_1 y_7, \text{ where} \\
s &:= \mathcal{R}\, x\, t_0\, (\lambda_x \lambda_p \textbf{let } x_p := p \textbf{ in } t_1), \\
t_0 &:= \lambda_{y_0} \textbf{let } y := y_0 \textbf{ in let } y_1 := y \textbf{ in } y_1, \\
t_1 &:= \lambda_{y_2} \textbf{let } f_0 := (\lambda_{y_3} \textbf{let } z := y_3 \textbf{ in let } y_4 := z + z \textbf{ in } y_4) \textbf{ in} \\
&\qquad\quad \textbf{let } y := y_2 \textbf{ in let } y_5 := y + 1 \textbf{ in let } z_0 := x_p y_5 \textbf{ in } f_0 z_0.
\end{aligned}
$$

Affine reductions

Let $\#_x(t)$ denote the number of free occurrences of x in t. Consider

$$(\lambda_x s)t \quad \mapsto_a \quad s\,[x := t]\,, \qquad \text{if } \#_x(t) \leq 1$$

Theorem

$\mapsto_a$ is strongly normalising and confluent.

Exponential example after affine reductions:

$$R := \lambda_x \mathcal{R}\, x\, (\lambda_y y)\ \left(\lambda_x \lambda_p \lambda_y (\lambda_z z + z)(p(y+1))\right)$$

Affine reductions

Let $\#_x(t)$ denote the number of free occurrences of x in t. Consider

$$(\lambda_x s)t \quad \mapsto_a \quad s\,[x := t]\,, \qquad \text{if } \#_x(t) \leq 1$$

Theorem

$\mapsto_a$ is strongly normalising and confluent.

Exponential example after affine reductions:

$$R := \lambda_x \mathcal{R}\, x\, (\lambda_y y)\ \left(\lambda_x \lambda_p \lambda_y (\lambda_z z + z)(p(y+1))\right)$$

Affine reductions

Let $\#_x(t)$ denote the number of free occurrences of x in t. Consider

$$(\lambda_x s)t \quad \mapsto_a \quad s\,[x := t]\,, \qquad \text{if } \#_x(t) \leq 1$$

Theorem

$\mapsto_a$ is strongly normalising and confluent.

Exponential example after affine reductions:

$$R := \lambda_x \mathcal{R}\, x\, (\lambda_y y)\ \left(\lambda_x \lambda_p \lambda_y (\lambda_z z + z)(p(y+1))\right)$$

Recursive uniformisation

Let $R \subseteq \mathbb{N} \times \mathbb{N}$ be recursive with $dom(R) = \mathbb{N}$.
Then there is a recursive function *uniformising* R.

We consider initial approximations of a uniformising function.

$$\forall_x \exists_y R(x, y) \to \forall_n \exists_l \left(|l| = n \wedge \forall_m (m < n \to R(n - m - 1, l_m)) \right)$$

Recursive uniformisation

Let $R \subseteq \mathbb{N} \times \mathbb{N}$ be recursive with $dom(R) = \mathbb{N}$.
Then there is a recursive function *uniformising* R.

We consider initial approximations of a uniformising function.

$$\forall_x \exists_y R(x, y) \to \forall_n \exists_l \big(|l| = n \wedge \forall_m (m < n \to R(n - m - 1, l_m))\big)$$

Extracted programs

$$\forall_x \tilde{\exists}_y R(x, y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall_m (m < n \to R(n - m - 1, l_m))\big)$$

$$\begin{aligned}
t &:= \langle t_+, t_- \rangle \\
t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g \mathrm{nil})\, (\lambda_{n,p,g} (fn) :: ph) \\
t_- &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g 0)\, (\lambda_{n,p,g} \textbf{if } R(n, fn) \textbf{ then } ph \textbf{ else } n) \\
h &:= \lambda_l \textbf{ if } g((fn) :: l) = 0 \textbf{ then } 0 \textbf{ else } g((fn) :: l) - 1
\end{aligned}$$

Extracted programs

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall_m (m < n \to R(n-m-1, l_m)) \big)$$

$$\begin{aligned}
t &:= \langle t_+, t_- \rangle \\
t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g \mathrm{nil})\, (\lambda_{n,p,g} (fn) :: ph) \\
t_- &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g 0)\, (\lambda_{n,p,g} \textbf{if } R(n, fn) \textbf{ then } ph \textbf{ else } n) \\
h &:= \lambda_l \textbf{ if } g((fn) :: l) = 0 \textbf{ then } 0 \textbf{ else } g((fn) :: l) - 1
\end{aligned}$$

Legend:

- t_+ — realiser for $\tilde{\exists}_l$
- t_- — challenge for $\forall_x$
- $f : \mathbb{N} \Rightarrow \mathbb{N}$ — realising function for $\forall_x \tilde{\exists}_y$
- $g, h : \mathbb{L}(\mathbb{N}) \Rightarrow \mathbb{N}$ — challenging functions for $\forall_m$

Extracted programs

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned}
t &:= \langle t_+, t_- \rangle \\
t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g \mathrm{nil})\, (\lambda_{n,p,g} (fn) :: ph) \\
t_- &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g 0)\, (\lambda_{n,p,g} \textbf{if } R(n, fn) \textbf{ then } ph \textbf{ else } n) \\
h &:= \lambda_l \textbf{ if } g((fn) :: l) = 0 \textbf{ then } 0 \textbf{ else } g((fn) :: l) - 1
\end{aligned}$$

Legend:

- t_+ — realiser for $\tilde{\exists}_l$
- t_- — challenge for $\forall_x$
- $f : \mathbb{N} \Rightarrow \mathbb{N}$ — realising function for $\forall_x \tilde{\exists}_y$
- $g, h : \mathbb{L}(\mathbb{N}) \Rightarrow \mathbb{N}$ — challenging functions for $\forall_m$

Extracted programs

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall_m (m < n \to R(n-m-1, l_m)) \big)$$

$$\begin{aligned}
t &:= \langle t_+, t_- \rangle \\
t_+ &:= \lambda_{f,n} \mathcal{R}\, n \, (\lambda_g \mathrm{nil}) \, (\lambda_{n,p,g} (fn) :: ph) \\
t_- &:= \lambda_{f,n} \mathcal{R}\, n \, (\lambda_g 0) \, (\lambda_{n,p,g} \textbf{if } R(n, fn) \textbf{ then } ph \textbf{ else } n) \\
h &:= \lambda_l \textbf{ if } g((fn) :: l) = 0 \textbf{ then } 0 \textbf{ else } g((fn) :: l) - 1
\end{aligned}$$

Legend:

- t_+ — realiser for $\tilde{\exists}_l$
- t_- — challenge for $\forall_x$
- $f : \mathbb{N} \Rightarrow \mathbb{N}$ — realising function for $\forall_x \tilde{\exists}_y$
- $g, h : \mathbb{L}(\mathbb{N}) \Rightarrow \mathbb{N}$ — challenging functions for $\forall_m$

Extracted programs

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \mathbin{\tilde{\wedge}} \forall_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned} t &:= \langle t_+, t_- \rangle \\ t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g \mathrm{nil})\, (\lambda_{n,p,g} (fn) :: ph) \\ t_- &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g 0)\, (\lambda_{n,p,g} \textbf{if } R(n, fn) \textbf{ then } ph \textbf{ else } n) \\ h &:= \lambda_l \textbf{ if } g((fn) :: l) = 0 \textbf{ then } 0 \textbf{ else } g((fn) :: l) - 1 \end{aligned}$$

Legend:

- t_+ — realiser for $\tilde{\exists}_l$
- t_- — challenge for $\forall_x$
- $f : \mathbb{N} \Rightarrow \mathbb{N}$ — realising function for $\forall_x \tilde{\exists}_y$
- $g, h : \mathtt{L}(\mathbb{N}) \Rightarrow \mathbb{N}$ — challenging functions for $\forall_m$

Computational uniformities

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned}
t &:= \langle t_+, t_- \rangle \\
t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g \mathrm{nil})\, (\lambda_{n,p,g} (fn) :: ph) \\
t_- &:= \lambda_{f,n} \mathcal{R}\, n\, (\lambda_g 0)\, (\lambda_{n,p,g} \textbf{if } R(n, fn) \textbf{ then } ph \textbf{ else } n) \\
h &:= \lambda_l \textbf{ if } g((fn) :: l) = 0 \textbf{ then } 0 \textbf{ else } g((fn) :: l) - 1
\end{aligned}$$

The functions g and h are computationally irrelevant!

Computational uniformities

$$\forall_x \tilde{\exists}_y R(x, y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall_m (m < n \to R(n - m - 1, l_m)) \big)$$

$$\begin{aligned}
t &:= \langle t_+, t_- \rangle \\
t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, \mathrm{nil}\, (\lambda_{n,p} (fn) :: p) \\
t_- &:= \lambda_{f,n} \mathcal{R}\, n\, 0\, (\lambda_{n,p} \textbf{if } R(n, fn) \textbf{ then } p \textbf{ else } n)
\end{aligned}$$

The functions g and h are computationally irrelevant!

Computational uniformities

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned} t &:= \langle t_+, t_- \rangle \\ t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, \mathrm{nil}\, (\lambda_{n,p}(fn) :: p) \\ t_- &:= \lambda_{f,n} \mathcal{R}\, n\, 0\, (\lambda_{n,p} \textbf{if } R(n, fn) \textbf{ then } p \textbf{ else } n) \end{aligned}$$

The functions g and h are computationally irrelevant!

$\forall_m$ is computationally uniform.

Uniform annotations

Definition

A proof is *uniformly interpretable* if for every needed case distinction on a formula C it has no uniform annotations.

Definition

A uniformly interpretable proof is *computationally correct* if

rule	flags	restriction
$\lambda_x M$	$\overset{-}{\forall}$	$x \notin \bigcup \mathrm{FV}(\{\!\vert M\vert\!\}_i^-)$
	$\overset{\pm}{\forall}$	$x \notin \mathrm{FV}(\{\!\vert M\vert\!\})$
$\lambda_{u_0} M$	$\xrightarrow{-\quad}$	$x_0 \notin \bigcup \mathrm{FV}(\{\!\vert M\vert\!\}_i^-)$
	$\xrightarrow{\quad -}$	$y \notin \bigcup \mathrm{FV}(\{\!\vert M\vert\!\}_i^- y)$
	$\xrightarrow{-\ -}$	$x_0, y \notin \bigcup \mathrm{FV}(\{\!\vert M\vert\!\}_i^- y)$
	$\xrightarrow{\pm\quad}$	$x_0 \notin \mathrm{FV}(\{\!\vert M\vert\!\})$
	$\xrightarrow{\pm\ -}$	$x_0 \notin \mathrm{FV}(\{\!\vert M\vert\!\})$ $y \notin \bigcup \mathrm{FV}(\{\!\vert M\vert\!\}_i^- y)$

Uniform annotations

Definition

A proof is *uniformly interpretable* if for every needed case distinction on a formula C it has no uniform annotations.

Definition

A uniformly interpretable proof is *computationally correct* if

rule	flags	restriction
$\lambda_x M$	$\overset{-}{\forall}$	$x \notin \bigcup \mathrm{FV}(\{\!\|M\|\!\}_i^-)$
	$\overset{\pm}{\forall}$	$x \notin \mathrm{FV}(\{\!\|M\|\!\})$
$\lambda_{u_0} M$	$\xrightarrow{-}$	$x_0 \notin \bigcup \mathrm{FV}(\{\!\|M\|\!\}_i^-)$
	$\xrightarrow{\quad -}$	$y \notin \bigcup \mathrm{FV}(\{\!\|M\|\!\}_i^- y)$
	$\xrightarrow{- \ -}$	$x_0, y \notin \bigcup \mathrm{FV}(\{\!\|M\|\!\}_i^- y)$
	$\xrightarrow{\pm}$	$x_0 \notin \mathrm{FV}(\{\!\|M\|\!\})$
	$\xrightarrow{\pm \ -}$	$x_0 \notin \mathrm{FV}(\{\!\|M\|\!\})$ $y \notin \bigcup \mathrm{FV}(\{\!\|M\|\!\}_i^- y)$

Avoiding recomputations

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m)) \big)$$

$$\begin{aligned} t &:= \langle t_+, t_- \rangle \\ t_+ &:= \lambda_{f,n} \mathcal{R}\, n\, \mathrm{nil}\, (\lambda_{n,p} (fn) :: p) \\ t_- &:= \lambda_{f,n} \mathcal{R}\, n\, 0\, (\lambda_{n,p} \textbf{if } R(n, fn) \textbf{ then } p \textbf{ else } n) \end{aligned}$$

t_+ and t_- are calculated using the same recursive scheme.

Avoiding recomputations

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m)) \big)$$

$$\begin{aligned} t \quad := \lambda_{f,n} \mathcal{R}\, n\, \langle \mathrm{nil}, 0 \rangle\ (\lambda_{n,p_l,p_n} &\textbf{let } m := fn \textbf{ in} \\ &\langle m :: p_l, \textbf{if } R(n,m) \textbf{ then } p_n \textbf{ else } n \rangle) \end{aligned}$$

t_+ and t_- are calculated using the same recursive scheme.

We can pack the two terms into a single computation.

Optimising recursion

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned} t \quad := \lambda_{f,n} \mathcal{R}\, n\, \langle \mathrm{nil}, 0 \rangle\ (\lambda_{n,p_l,p_n} &\textbf{let } m := fn \textbf{ in} \\ &\langle m :: p_l, \textbf{if } R(n,m) \textbf{ then } p_n \textbf{ else } n \rangle) \end{aligned}$$

Can we optimise further?

- positive computation is optimal
- negative computation searches for the last failure index
- it is sufficient to stop at the first failure index
- we mark successful counterexample candidates as $n \blacktriangleright \mathrm{ff}$
- and skip further checks if they are redundant

Optimising recursion

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathrm{U}}_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned} t \;\; &:= \lambda_{f,n} \mathcal{R}\, n \,\langle \mathrm{nil}, 0\rangle\; (\lambda_{n,p_l,p_n} \textbf{let } m := fn \textbf{ in} \\ &\qquad\qquad\qquad\qquad \langle m :: p_l, \textbf{if } R(n,m) \textbf{ then } p_n \textbf{ else } n\rangle) \end{aligned}$$

Can we optimise further?

- positive computation is optimal
- negative computation searches for the last failure index
- it is sufficient to stop at the first failure index
- we mark successful counterexample candidates as $n \blacktriangleright \mathrm{ff}$
- and skip further checks if they are redundant

Optimising recursion

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l (\, |l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m)))$$

$$\begin{aligned} t \;&:= \lambda_{f,n} \mathcal{R}\, n\, \langle \mathrm{nil}, 0 \rangle\; (\lambda_{n,p_l,p_n} \textbf{let } m := fn \textbf{ in} \\ &\qquad\qquad\qquad\qquad \langle m :: p_l, \textbf{if } R(n,m) \textbf{ then } p_n \textbf{ else } n \rangle) \end{aligned}$$

Can we optimise further?

- positive computation is optimal
- negative computation searches for the last failure index
- it is sufficient to stop at the first failure index
- we mark successful counterexample candidates as $n \blacktriangleright \mathrm{ff}$
- and skip further checks if they are redundant

Optimising recursion

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned} t \ &:= \lambda_{f,n} \mathcal{R}\, n\, \langle \mathrm{nil}, 0 \blacktriangleright \mathrm{tt} \rangle\ \big(\lambda_{n,p_l,p_n \blacktriangleright p_b} \textbf{let } m := fn \textbf{ in} \\ &\qquad \langle m :: p_l, \textbf{if } \neg p_b \vee R(n,m) \textbf{ then } p_n \blacktriangleright p_b \textbf{ else } n \blacktriangleright \mathrm{ff} \rangle \big) \end{aligned}$$

Can we optimise further?

- positive computation is optimal
- negative computation searches for the last failure index
- it is sufficient to stop at the first failure index
- we mark successful counterexample candidates as $n \blacktriangleright \mathrm{ff}$
- and skip further checks if they are redundant

Optimising recursion

$$\forall_x \tilde{\exists}_y R(x,y) \to \forall_n \tilde{\exists}_l \big(|l| = n \,\tilde{\wedge}\, \forall^{\mathsf{U}}_m (m < n \to R(n-m-1, l_m))\big)$$

$$\begin{aligned} t \quad &:= \lambda_{f,n} \mathcal{R}\, n \,\langle \mathrm{nil}, 0 \blacktriangleright \mathrm{tt}\rangle \, \big(\lambda_{n,p_l,p_n \blacktriangleright p_b} \textbf{let } m := fn \textbf{ in} \\ &\qquad\qquad \langle m :: p_l, \textbf{if } \neg p_b \vee R(n,m) \textbf{ then } p_n \blacktriangleright p_b \textbf{ else } n \blacktriangleright \mathrm{ff}\rangle\big) \end{aligned}$$

Can we optimise further?

- positive computation is optimal
- negative computation searches for the last failure index
- it is sufficient to stop at the first failure index
- we mark successful counterexample candidates as $n \blacktriangleright \mathrm{ff}$
- and skip further checks if they are redundant

Counterexample marking

- $t \blacktriangleright \mathrm{tt}$ — we have no information about the validity of $(\!| C_i |\!)_t^{x_i}$,
- $t \blacktriangleright \mathrm{ff}$ — we have checked that $\neg(\!| C_i |\!)_t^{x_i}$,

Counterexample marking

- $t \blacktriangleright \mathrm{tt}$ — we have no information about the validity of $(\!| C_i |\!)_t^{x_i}$,
- $t \blacktriangleright \mathrm{ff}$ — we have checked that $\neg(\!| C_i |\!)_t^{x_i}$,

Counterexample marking

- $t \blacktriangleright \mathrm{tt}$ — we have no information about the validity of $(\!|C_i|\!)_t^{x_i}$,
- $t \blacktriangleright \mathrm{ff}$ — we have checked that $\neg(\!|C_i|\!)_t^{x_i}$,

Lemma

For any formula in NA^ω *and let* $x : \rho^*(C)$ *be a variable. Then there is a term* $T^C_{\bowtie} : \rho^{\multimap}(C) \Rightarrow \rho^{\multimap}(C) \Rightarrow \rho^{\multimap}(C)$ *with* $\mathrm{FV}(T^C_{\bowtie}) \subseteq \mathrm{FV}(C) \cup \{x\}$*, such that for* $t_1, t_2 : \rho^{\multimap}(C)$

A_i : $(\!|C|\!)_s^x \to (\!|C|\!)_{s_i}^x$,

B : $(\!|C|\!)_s^x \to \mathrm{at}(m)$,

where $t_i := s_i \blacktriangleright m_i$ *and* $s \blacktriangleright m = T^C_{\bowtie}\, t_1\, t_2$.

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity (“abort” effect)

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity ("abort" effect)

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity ("abort" effect)

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity ("abort" effect)

Future work

- implement optimizations in Minlog
- experiment with larger case studies
- find other general improvements

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity (“abort” effect)

Future work

- implement optimizations in Minlog
- experiment with larger case studies
- find other general improvements

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity (“abort” effect)

Future work

- implement optimizations in Minlog
- experiment with larger case studies
- find other general improvements

Conclusion and future work

The original Dialectica intepretation can be modified in a sound way to produce better programs:

- programs are shorter (no code repetition, no redundant code)
- better worst time complexity (no recomputation, no redundant code)
- better average time complexity (“abort” effect)

Future work

- implement optimizations in Minlog
- experiment with larger case studies
- find other general improvements

Thank you

Thank you for your attention!