Lecture 12
Synthesis from Relations and Types

Viktor Kuncak

Synthesis of Functions from Relations

- We previously saw: how to convert programs into formulas
  - verification-condition generation
  - formulas describe relation between inputs and outputs
- Now: convert formulas into programs (opposite direction)

Example domain: Presburger arithmetic

Programs and Specs are Relations

$$\begin{array}{ll} \text{program:} & x = x + 2; y = x + 10 \\ \text{relation:} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \text{formula:} & x' = x + 2 \wedge y' = x + 12 \wedge z' = z \end{array}$$

Specification:

$$z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0)$$

Adhering to specification is relation subset:

$$\begin{array}{ll} & \{(x, y, z, x', y', z') \mid x' = x + 2 \wedge y' = x + 12 \wedge z' = z\} \\ \subseteq & \{(x, y, z, x', y', z') \mid z' = z \wedge (x > 0 \rightarrow (x' > 0 \wedge y' > 0))\} \end{array}$$

Non-deterministic programs are a way of writing specifications

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Use local variables to store initial values.

Writing Specs Using Havoc and Assume: Examples

Program variables $V = \{x, y, z\}$

Formula for relation (talks only about resulting state):

$$z' = z \wedge x' > 0 \wedge y' > 0$$

Corresponding program:

$$havoc(x, y); assume(x > 0 \wedge y > 0)$$

Formula for relation:

$$z' = z \wedge x' > x \wedge y' > y$$

Corresponding program?

Use local variables to store initial values.

```
{ var x0; var y0;
  x0 = x; y0 = y;
  havoc(x,y);
  assume(x > x0 && y > y0)
}
```

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

Writing Specs Using Havoc and Assume

Global variables $V = \{x_1, \ldots, x_n\}$
Specification

$$F(x_1, \ldots, x_n, x_1', \ldots, x_n')$$

Becomes

$$\begin{array}{l} \{\ \mathit{var}\ y_1, \ldots, y_n; \\ \quad y_1 = x_1; \ldots; y_n = x_n; \\ \quad \mathit{havoc}(x_1, \ldots, x_n); \\ \quad \mathit{assume}(F(y_1, \ldots, y_n, x_1, \ldots, x_n))\ \} \end{array}$$

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.
(Some books use the opposite meaning of $\sqsubseteq$.)
As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{\mathit{var}\ x0; x0 = x; \mathit{havoc}(x); \mathit{assume}(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

Program Refinement and Equivalence

For two programs, define **refinement** $P_1 \sqsubseteq P_2$ iff

$$R(P_1) \rightarrow R(P_2)$$

is a valid formula.
(Some books use the opposite meaning of $\sqsubseteq$.)
As usual, $P_2 \sqsupseteq P_1$ iff $P_1 \sqsubseteq P_2$.

- $P_1 \sqsubseteq P_2$ iff $\rho(P_1) \subseteq \rho(P_2)$

Define **equivalence** $P_1 \equiv P_2$ iff $P_1 \sqsubseteq P_2 \wedge P_2 \sqsubseteq P_1$

- $P_1 \equiv P_2$ iff $\rho(P_1) = \rho(P_2)$

Example for $V = \{x, y\}$

$$\{var\ x0; x0 = x; havoc(x); assume(x > x0)\} \sqsupseteq (x = x + 1)$$

Proof: Use R to compute formulas for both sides and simplify.

$$x' = x + 1 \wedge y' = y \ \rightarrow \ x' > x \wedge y' = y$$

Stepwise Refinement Methodology

Start form a possibly non-deterministic specification P_0
Refine the program until it becomes deterministic and efficiently executable.

$$P_0 \sqsupseteq P_1 \sqsupseteq \ldots \sqsupseteq P_n$$

Example:

$$\begin{aligned}
& \mathit{havoc}(x); \mathit{assume}(x > 0); \mathit{havoc}(y); \mathit{assume}(x < y) \\
\sqsupseteq\ & \mathit{havoc}(x); \mathit{assume}(x > 0); y = x + 1 \\
\sqsupseteq\ & x = 42; y = x + 1 \\
\sqsupseteq\ & x = 42; y = 43
\end{aligned}$$

In the last step program equivalence holds as well

Preserving Domain in Refinement

What is the domain of a relation?

Given relation $r \subseteq A \times B$ for any sets A, B, we define domain of r as

$$dom(r) = \{a \mid \exists b.\ (a, b) \in r\}$$

when r is a total function, then $dom(r) = A$

- a typical case if r is an entire program

Let $r = \{(\bar{x}, \bar{x}') \mid F\}$, $FV(F) \subseteq Var \cup Var'$, $Var' = \{x' \mid x \in Var\}$. Then, $dom(r) = \{\bar{x} \mid \exists \bar{x}'.F\}$

- computing domain = existentially quantifying over primed vars

Example: for $Var = \{x, y\}$, $R(x = x + 1) = x' = x + 1 \wedge y' = y$. The formula for the domain is: $\exists x', y'.\ x' = x + 1 \wedge y' = y$, which, after one-pint rule, reduces to true.

- All assignments have true as domain.

Preserving Domain

It is not interesting program development step $P \sqsupseteq P'$ is P' is false, or is false for most inputs.
Example ($Var = \{x, y\}$)

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

Refinement $P \sqsupseteq Q$, ensures $R(Q) \rightarrow R(P)$. A consequence is $(\exists \bar{x}'.R(Q)) \rightarrow (\exists \bar{x}'.R(P))$.
We additionally wish to preserve the *domain* of the relation between $\bar{x}, \bar{x}'$

- if P has some execution from $\bar{x}$ ending in $\bar{x}'$
- then Q should also have some execution, ending in some (possibly different) $\bar{x}'$ (even if it has fewer choices)

$$(\exists \bar{x}'.R(P)) \leftrightarrow (\exists \bar{x}'.R(Q))$$

So, we want relations to be smaller or equal, but domains equal.

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(\mathit{havoc}(x); \mathit{assume}(x + x = y)\big) \quad \sqsupseteq \quad \big(\mathit{assume}(y = 6); x = 3\big)$$

- $R(P) =$

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') =$

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(\mathit{havoc}(x); \mathit{assume}(x + x = y)\big) \quad \sqsupseteq \quad \big(\mathit{assume}(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold?

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(\mathit{havoc}(x); \mathit{assume}(x + x = y)\big) \quad \sqsupseteq \quad \big(\mathit{assume}(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(\mathit{havoc}(x); \mathit{assume}(x + x = y)\big) \quad \sqsupseteq \quad \big(\mathit{assume}(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is $\exists x', y'.x' + x' = y \wedge y' = y$
- equivalent to:

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is $\exists x', y'.x' + x' = y \wedge y' = y$
- equivalent to: $y\%2 = 0$
- domain of P is:

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is $\exists x', y'.x' + x' = y \wedge y' = y$
- equivalent to: $y\%2 = 0$
- domain of P is: $\exists x', y'.x' = 3 \wedge y' = 6 \wedge y' = y$
- equivalent to:

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is $\exists x', y'.x' + x' = y \wedge y' = y$
- equivalent to: $y\%2 = 0$
- domain of P is: $\exists x', y'.x' = 3 \wedge y' = 6 \wedge y' = y$
- equivalent to: $y = 6$

Does domain formula of P' imply the domain formula of P?

Domains in the Example

Consider our example $P \sqsupseteq P'$

$$\big(havoc(x); assume(x + x = y)\big) \quad \sqsupseteq \quad \big(assume(y = 6); x = 3\big)$$

- $R(P) = x' + x' = y' \wedge y' = y$
- $R(P') = x' = 3 \wedge y' = 6 \wedge y' = y$

Does $P \sqsupseteq P'$ really hold? yes

Now consider the right hand side:

- domain of P is $\exists x', y'.x' + x' = y \wedge y' = y$
- equivalent to: $y\%2 = 0$
- domain of P is: $\exists x', y'.x' = 3 \wedge y' = 6 \wedge y' = y$
- equivalent to: $y = 6$

Does domain formula of P' imply the domain formula of P? no

Preserving Domain: Exercise

Given P:

$$havoc(x); assume(x + x = y)$$

Find P_1 and P_2 such that

- $P \sqsupseteq P_1 \sqsupseteq P_2$
- no two programs among P, P_1, P_2 are equivalent
- programs P, P_1 and P_2 have equivalent domains
- the relation described by P_2 is a partial function

Synthesis from Relations

Software Synthesis Procedures
Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, Philippe Suter
Communications of the ACM, Vol. 55 No. 2, Pages 103-111
http://doi.org/10.1145/2076450.2076472

Example of Synthesis

Input:

```
val (hours, minutes, seconds) = choose((h: Int, m: Int, s: Int) => (
  h * 3600 + m * 60 + s == totsec
  && 0 <= m && m < 60
  && 0 <= s && s < 60))
```

Output:

```
val (hours, minutes, seconds) = {
val loc1 = totsec div 3600
val num2 = totsec + ((-3600) * loc1)
val loc2 = min(num2 div 60, 59)
val loc3 = totsec + ((-3600) * loc1) + (-60 * loc2)
  (loc1, loc2, loc3)
}
```

Complete Functional Synthesis

Domain-preserving refinement algorithm that produces a partial function

- assignment: **res** = **choose x. F**
- corresponds to: $\{\textbf{var x}; \textbf{assume}(\textbf{F}); \textbf{res} = \textbf{x}\}$
- we refine it preserving domain into: $\textbf{assume}(\textbf{D}); \textbf{res} = \textbf{t}$ (where t does not have 'choose')

More abstractly, given formula F and variable x find

- formula D
- term t not containing x

such that, for all free variables:

- $D \rightarrow F[x := t]$ (t is a term such that refinement holds)
- $D \iff \exists x.F$ (D is the domain, says when t is correct)

Consequence of the definition: $D \iff F[x := t]$

From Quantifier Elimination to Synthesis

Quantifier Elimination

If $\bar{y}$ is a tuple of variables not containing x, then

$$\exists x.(x = t(\bar{y}) \wedge F(x, \bar{y})) \iff F(t(\bar{y}), \bar{y})$$

Synthesis

$$\mathit{choose}\ x.(x = t(\bar{y}) \wedge F(x, \bar{y}))$$

gives:

- precondition $F(t(\bar{y}), \bar{y})$, as before, but also
- program that realizes x, in this case, $t(\bar{y})$

Handling Disjunctions

We had

$$\exists x.(F_1(x) \vee F_2(x))$$

is equivalent to

$$(\exists x.F_1(x)) \vee (\exists x.F_2(x))$$

Now:

$$\mathit{choose}\ x.(F_1(x) \vee F_2(x))$$

becomes:

$$\mathit{if}\ (D_1)\ (\mathit{choose}\ x.F_1(x))\ \mathit{else}\ (\mathit{choose}\ x.F_2(x))$$

where D_1 is the domain, equivalent to $\exists x.F_1(x)$ and computed while computing $\mathit{choose}\ x.F_1(x)$.

Framework for Synthesis Procedures

We define the framework as a transformation

- from specification formula F to
- the maximal domain D where the result x can be found, and the program t that computes the result

$\langle D \mid t \rangle$ denotes: the domain (formula) D and program (term) t
Main transformation relation $\vdash$

$$\mathit{choose}\ x.F \ \vdash\ \langle D \mid t \rangle$$

means

- $D \rightarrow F[x := t]$ (t is a term such that refinement holds)
- $D \iff \exists x.F$ (D is the domain, says when t is correct)

Because $F[x := t]$ implies $\exists x.F$, the above definition implies that D, $F[x := t]$ and $\exists x.F$ are all equivalent.

Rule for Synthesizing Conditionals

$$\frac{\mathit{choose}\ x.F_1 \vdash \langle D_1 \mid t_1 \rangle \qquad \mathit{choose}\ x.F_2 \vdash \langle D_2 \mid t_2 \rangle}{\mathit{choose}\ x.(F_1 \vee F_2) \ \vdash\ \langle D_1 \vee D_2 \mid \mathit{if}\ (D_1)\ t_1\ \mathit{else}\ t_2 \rangle}$$

To synthesize the thing below the — , synthesize the things above and put the pieces together.

Test Terms Methods for Presburger Arithmetic Synthesis

Recall that the most complex step in QE for PA was replacing

$$\exists x.F_1(x)$$

with

$$\bigvee_{k=1}^{L} \bigvee_{i=1}^{N} F_1(a_k + i)$$

Now we transform *choose* $x.F_1(x)$ first into:

$$\textit{choose } x. \bigvee_{k=1}^{L} \bigvee_{i=1}^{N} (x = a_k + i \wedge F_1(x))$$

Then apply:

Test Terms Methods for Presburger Arithmetic Synthesis

Recall that the most complex step in QE for PA was replacing

$$\exists x.F_1(x)$$

with

$$\bigvee_{k=1}^{L} \bigvee_{i=1}^{N} F_1(a_k + i)$$

Now we transform *choose* $x.F_1(x)$ first into:

$$\textit{choose } x. \bigvee_{k=1}^{L} \bigvee_{i=1}^{N} (x = a_k + i \wedge F_1(x))$$

Then apply:

- rule for conditionals

Test Terms Methods for Presburger Arithmetic Synthesis

Recall that the most complex step in QE for PA was replacing

$$\exists x.F_1(x)$$

with

$$\bigvee_{k=1}^{L} \bigvee_{i=1}^{N} F_1(a_k + i)$$

Now we transform *choose* $x.F_1(x)$ first into:

$$\mathit{choose}\ x. \bigvee_{k=1}^{L} \bigvee_{i=1}^{N} (x = a_k + i \wedge F_1(x))$$

Then apply:

- rule for conditionals
- one-point rule

Synthesis using Test Terms

$$\mathit{choose}\ x.\ \bigvee_{k=1}^{L}\ \bigvee_{i=1}^{N}(x = a_k + i \wedge F_1)$$

produces the same precondition as the result of QE, and the generated term is:

$$\begin{array}{l}
\mathit{if}\ (F_1[x := a_1 + 1])\ a_1 + 1 \\
\mathit{elseif}\ (F_1[x := a_1 + 2])\ a_1 + 2 \\
\ldots \\
\mathit{elseif}\ (F_1[x := a_k + i])\ a_k + i \\
\ldots \\
\mathit{elseif}\ (F_1[x := a_L + N])\ a_L + N
\end{array}$$

Linear search over the possible values, taking the first one that works.
This could be optimized in many cases.

Synthesizing a Tuple of Outputs

$$\frac{\mathit{choose}\ x.F\ \vdash\ \langle D_1 \mid t_1 \rangle \quad \mathit{choose}\ y.D_1\ \vdash\ \langle D_2 \mid t_2 \rangle}{\mathit{choose}\ (x, y).F\ \vdash\ \langle D_2 \mid (t_1[y := t_2],\ t_2) \rangle}$$

Note that y can appear inside D_1 and t_1, but not in D_2 or t_2

Substitution of Variables

In quantifier elimination, we used a step where we replace $M \cdot x$ with y. Let F be a formula in which x occurs only in the form $M \cdot x$.

What is the corresponding rule?

Substitution of Variables

In quantifier elimination, we used a step where we replace $M \cdot x$ with y. Let F be a formula in which x occurs only in the form $M \cdot x$.
What is the corresponding rule?

$$\frac{\mathit{choose}\ y.(F[(M \cdot x) := y] \wedge (M|y)) \ \vdash\ \langle D \mid t \rangle}{\mathit{choose}\ x.F \ \vdash\ \langle D \mid t[y := t/M] \rangle}$$

Automated Checks for Specifications: Uniqueness

Suppose we wish to give a warning if the specification F allows two different solutions.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$\mathit{choose}\ x.\ F$$

What is the verification condition that checks whether the solution for x is unique?

Automated Checks for Specifications: Uniqueness

Suppose we wish to give a warning if the specification F allows two different solutions.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$\mathit{choose}\ x.\ F$$

What is the verification condition that checks whether the solution for x is unique?
Solution is **not** unique if this PA formula is satisfiable:

Automated Checks for Specifications: Uniqueness

Suppose we wish to give a warning if the specification F allows two different solutions.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$\mathit{choose}\ x.\ F$$

What is the verification condition that checks whether the solution for x is unique?
Solution is **not** unique if this PA formula is satisfiable:

$$F \wedge F[x := y] \wedge x \neq y$$

If we find such x, y, z we report z as an example input for which there are two possible outputs, x and y.

Automated Checks for Specifications: Totality

Suppose we wish to give a warning if in some cases the solution does not exist.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$\mathit{choose}\ x.\ F$$

What is the verification condition that checks if there are cases when no solution x exists?

Automated Checks for Specifications: Totality

Suppose we wish to give a warning if in some cases the solution does not exist.

Let the variables in scope be denoted by z and consider the synthesis problem:

$$\mathit{choose}\ x.\ F$$

What is the verification condition that checks if there are cases when no solution x exists?
Check satisfiability of this PA formula:

$$\neg\exists x.F$$

If there is a satisfying value for this formula, z, report it as an example for which no solution for x exists.

Synthesis of Functions of a Given Type

Type Judgments and Questions

In environment Γ, expression e has type T:

$$\Gamma \vdash e : T$$

After defining this relation inductively using type rules, we can ask different types of questions:

		given	task
type checking	$\Gamma \vdash e : T$	Γ, e, T	check if $\Gamma \vdash e : T$
type inference	$\Gamma \vdash e : ?$	Γ, e	find T s.t. $\Gamma \vdash e : T$
type inhabitation	$\Gamma \vdash ? : T$	Γ, T	find e s.t. $\Gamma \vdash e : T$

Inhabitation can be used for type-directed completion: in a given program context, can we construct the expression of a given type?

Also, type T can be viewed as a proposition, program e as a proof. Inhabitation asks if the proposition T has some proof e (if it is a theorem, usually in some constructive logic).

Simply Typed Lambda Calculus (Church Style)

ee_1 means e applied to e_1; in Scala: $e(e_1)$

$\lambda x : T_1.e$ is anonymous function, in Scala: $(x : T_1) \Rightarrow e$

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash ee_1 : T_2} \quad \text{application}$$

$$\frac{\Gamma[x := T_1] \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.\ e_2) : T_1 \to T_2} \quad \text{abstraction}$$

Type Checking is Easy and Types Unique

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash e e_1 : T_2} \quad \text{application}$$

$$\frac{\Gamma[x := T_1] \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : \mathbf{T_1}.\ e_2) : T_1 \to T_2} \quad \text{abstraction}$$

Which Cases of Inhabitation are Difficult?

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash e e_1 : T_2} \quad \text{application}$$

$$\frac{\Gamma[x := T_1] \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.\ e_2) : T_1 \to T_2} \quad \text{abstraction}$$

Suppose that we are equally happy with any of the possible terms of a given type.

Which Cases of Inhabitation are Difficult?

$$\frac{(x, T) \in \Gamma}{\Gamma \vdash x : T} \quad \text{variable}$$

$$\frac{\Gamma \vdash e : T_1 \to T_2, \quad \Gamma \vdash e_1 : T_1}{\Gamma \vdash e e_1 : T_2} \quad \text{application}$$

$$\frac{\Gamma[x := T_1] \vdash e_2 : T_2}{\Gamma \vdash (\lambda x : T_1.\ e_2) : T_1 \to T_2} \quad \text{abstraction}$$

Suppose that we are equally happy with any of the possible terms of a given type.

Challenge: in application rule, T_1 can be arbitrarily complex type; we would need to guess it.

Approaches to Solve the Difficulty

For more complex type systems, the problem is undecidable

- we can choose to restrict our search to expressions e of some bounded size
- the problem becomes decidable if type checking is decidable: can try all terms up to given size

In our simple case of simply typed lambda calculus: there is a **terminating algorithm** that solves type inhabitation problem (in deterministic singly exponential time and polynomial space), without requiring any bound on the size of e.

Key Idea: Long Normal Form of Lambda Terms

Reductions preserve types of terms. Apply them to get normal form.

Beta reduction: $(\lambda x.e_1)e_2 \longrightarrow \text{subst}(e_1, x, e_2)$

If applied exhaustively ensures that left side of application is never a lambda.
This process can blow up, but terminates ("strong normalization").

Result: applications have the form $fe_1 \ldots e_n$ where f is a variable.

Key Idea: Long Normal Form of Lambda Terms

Eta expansion: when e has type $T_1 \to T_2$ then:
$e \longrightarrow (\lambda x : T_1 . e_1 x)$
Strategy: replace partially applied type variables by adding missing arguments.
Given

$$f : T_1 \to (T_2 \to \ldots (T_n \to T))$$

where T is not a function type, if $k < n$ then replace

$$f e_1 \ldots e_k$$

that is not of the form $f e_1 \ldots e_k e_{k+1}$ by

$$\lambda x_{k+1} . \lambda x_{k+2} . \ \ldots \lambda x_n . \ f e_1 \ldots e_k x_{k+1} \ldots x_n$$

As a result, application is always to a variable and applies all of its arguments that the type permits.

Type Rules for Long Normal Form Terms

full variable application ($n \geq 0$), if T is non-function type:

$$\frac{(f, T_1 \to \ldots \to T_n \to T) \in \Gamma, \quad \Gamma \vdash e_1 : T_1, \ \ldots \ \Gamma \vdash e_n : T_n}{\Gamma \vdash f e_1 \ldots e_n : T}$$

abstraction:

$$\frac{\Gamma \uplus \{(x_1, T_1), \ldots, (x_n, T_n) \ \vdash e : T}{\Gamma \vdash (\lambda x_1 : T_1. \ \ldots x_n : T_n . e) : T_1 \to \ldots \to T_n \to T}$$

- if there is a term of given type, this calculus derives some equivalent term of this type
- applying rules backwards generates queries with types that appear somewhere in environment or the query (possibly inside the syntactically larger types)
- there is no need to explore query $\Gamma \vdash ? : T$ if a query $\Gamma' \vdash ? : T$ was already asked for $dom(\Gamma) \subseteq dom(\Gamma')$

There is a finite number of extensions of the original Γ, so the search can stop after a finite amount of steps.

Using Inhabitation to Generate Suggestions

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In ACM SIGPLAN PLDI, 2013. `https://doi.org/10.1145/2499370.2462192`

- optimize representation to ignore duplicate arguments of the same type to check inhabitation (succinct terms)
- extend the algorithm to enumerate terms of given type
- instead of ordering terms by size, assign cost for each variable (cost of term is sum of its variables)
- map synthesis of Scala and Java code to lambda calculus
- statistically estimate the cost of variables from code corpus