1 DCC recap

1.1 Syntax

$$
\begin{array}{rll}
P & ::= \overline{D} & \text{(Program)} \\
D & ::= C(x.\ \overline{a}) \mid \forall x.\ \overline{a} \Rightarrow a \mid m(x.\ \overline{a}) : t \mid m(x.\ \overline{a}) : t := e & \text{(Decl)} \\
t & ::= [x.\ \overline{a}] & \text{(Type)} \\
a & ::= p \equiv p \mid p :: C \mid p.\mathbf{cls} \equiv C & \text{(Constr)} \\
p & ::= x \mid p.f & \text{(Path)} \\
e & ::= x \mid e.f \mid \mathbf{new}\ \ C(\overline{f} \equiv \overline{e}) & \text{(Expr)}
\end{array}
$$

1.2 Constraint Entailment

C-Ident
$$a \vdash a$$

C-Refl
$$\epsilon \vdash p \equiv p$$

C-Class
$$\frac{\overline{a} \vdash p.\mathbf{cls} \equiv C}{\overline{a} \vdash p :: C}$$

C-Cut
$$\frac{\overline{a} \vdash c \qquad \overline{a'}, c \vdash b}{\overline{a}, \overline{a'} \vdash b}$$

C-Subst
$$\frac{\overline{a} \vdash a_{\{\!\{x \mapsto p\}\!\}} \qquad \overline{a} \vdash p' \equiv p}{\overline{a} \vdash a_{\{\!\{x \mapsto p'\}\!\}}}$$

C-Prog
$$\frac{(\forall x.\ \overline{a} \Rightarrow a) \in P \qquad \overline{b} \vdash \overline{a}_{\{\!\{x \mapsto p\}\!\}}}{\overline{a} \vdash a_{\{\!\{x \mapsto p\}\!\}}}$$

1.3 Operational Semantics

R-New
$$\frac{\begin{array}{cc} x \notin dom(h) & o = \langle C; \overline{f \equiv x} \rangle \\ C(x.\ \overline{b}) \in P & HC(h), OC(x, o) \vdash \overline{b} \end{array}}{\langle h; \mathbf{new}\ \ C(\overline{f} \equiv \overline{x}) \rangle \rightarrow \langle h, x \mapsto o; x \rangle}$$

R-Field
$$\frac{(x.f \equiv y) \in HC(h)}{\langle h; x.f \rangle \rightarrow \langle h; y \rangle}$$

R-Call
$$\frac{\begin{array}{c} S = \{\langle \overline{a}; e \rangle | \langle \overline{a}; e \rangle \in mImpl(m, x) \wedge HC(h) \vdash \overline{a}\} \\ \langle \overline{a}; e \rangle \in S \qquad \forall \langle \overline{a'}; e' \rangle \in S.\ (e' \neq e) \longrightarrow \overline{a'} \vdash \overline{a} \wedge \neg \overline{a} \vdash \overline{a'} \end{array}}{\langle h; m(x) \rangle \rightarrow \langle h; e \rangle}$$

RC-Field
$$\frac{\langle h; e \rangle \rightarrow \langle h'; e' \rangle}{\langle h; e.f \rangle \rightarrow \langle h'; e'.f \rangle}$$

RC-Call
$$\frac{\langle h; e \rangle \rightarrow \langle h'; e' \rangle}{\langle h; m(e) \rangle \rightarrow \langle h'; m(e') \rangle}$$

RC-New
$$\frac{\langle h; e \rangle \rightarrow \langle h'; e' \rangle}{\langle h; \mathbf{new}\ \ C(\overline{f} \equiv \overline{x}, f \equiv e, \overline{f'} \equiv \overline{e'}) \rangle \rightarrow \langle h'; \mathbf{new}\ \ C(\overline{f} \equiv \overline{x}, f \equiv e', \overline{f'} \equiv \overline{e'}) \rangle}$$

1.4 Type Assignment

$$\textsc{T-Field}\quad \frac{\begin{array}{cc}\overline{c} \vdash e : [x.\ \overline{a}] & \overline{c}, \overline{a} \vdash x.f :: C \\ \overline{c}, \overline{a}, x.f \equiv y \vdash \overline{b} & x \notin FV(\overline{b})\end{array}}{\overline{c} \vdash e.f : [y.\ \overline{b}]}$$

$$\textsc{T-Call}\quad \frac{\begin{array}{ccc}\overline{c} \vdash e : [x.\ \overline{a}] & & \langle \overline{a'}; \overline{b} \rangle \in \mathit{MType}(m, x, y) \\ \overline{c}, \overline{a} \vdash \overline{a'} & \overline{c}, \overline{a}, \overline{b} \vdash \overline{b'} & x \notin FV(\overline{b'})\end{array}}{\overline{c} \vdash m(e) : [y.\ \overline{b'}]}$$

$$\textsc{T-Var}\quad \frac{\overline{c} \vdash x :: C}{\overline{c} \vdash x : [y.\ y \equiv x]}$$

$$\textsc{T-Sub}\quad \frac{\begin{array}{c}\overline{c} \vdash e : [x.\ \overline{a'}] \\ \overline{c}, \overline{a'} \vdash \overline{a}\end{array}}{\overline{c} \vdash e : [x.\ \overline{a}]}$$

$$\textsc{T-New}\quad \frac{\begin{array}{cc} \multicolumn{2}{c}{\forall i.\ \overline{c} \vdash e_i : [x_i.\ \overline{a_i}]} \\ \multicolumn{2}{c}{\overline{b} = (x.\mathbf{cls} \equiv C), \cup_i \overline{a_i}_{\{\!\{x_i \mapsto x.f_i\}\!\}}} \\ C(x.\ \overline{b'}) \in P & \overline{c}, \overline{b} \vdash \overline{b'}\end{array}}{\overline{c} \vdash \mathbf{new}\ C(\overline{f} \equiv \overline{e}) : [x.\ \overline{b}]}$$

1.5 Type Checking

$$\text{WF-CD}\quad \frac{FV(\overline{a}) = \{x\}}{\mathsf{wf}\, C(x.\ \overline{a})}$$

$$\text{WF-MS}\quad \frac{FV(\overline{a}) = \{x\} \qquad FV(\overline{b}) = \{x, y\}}{\mathsf{wf}\, m(x.\ \overline{a}) : [y.\ \overline{b}]}$$

$$\text{WF-RD}\quad \frac{FV(\overline{a}) = \{x\} \qquad x :: C' \in \overline{a}}{\mathsf{wf} \forall x.\ \overline{a} \Rightarrow x :: C}$$

$$\text{WF-MI}\quad \frac{FV(\overline{a}) = \{x\} \qquad FV(\overline{b}) = \{x, y\} \qquad \overline{a} \vdash e : [y.\ \overline{b}]}{\mathsf{wf}\, m(x.\ \overline{a}) : [y.\ \overline{b}] := e}$$

$$\textsc{WF-Prog}\quad \frac{\begin{array}{c}\forall D \in P.\ \mathsf{wf} D \qquad \forall m.\ \forall \langle \overline{a}; \overline{b} \rangle, \langle \overline{a'}; \overline{b'} \rangle \in \mathit{MType}(m, x, y).\ \overline{b} = \overline{b'} \\ \forall m.\ \forall \langle \overline{a}; \overline{b} \rangle \in \mathit{MType}(m, x, y).\ \text{complete}(m, [x.\ \overline{a}]) \qquad \forall m.\text{unique}(m)\end{array}}{\mathsf{wf} P}$$

1.6 Example Program

$$\begin{array}{l}
\texttt{Zero}(x.\ \epsilon) \\
\texttt{Succ}(x.\ x.p :: \texttt{Nat}) \\
\forall x.\ x :: \texttt{Zero} \Rightarrow x :: \texttt{Nat} \\
\forall x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat} \Rightarrow x :: \texttt{Nat} \\
\texttt{prev}(x.\ x :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}] \\
\texttt{prev}(x.\ x :: \texttt{Zero}) : [y.\ y :: \texttt{Nat}] := \mathbf{new}\ \texttt{Zero}() \\
\texttt{prev}(x.\ x :: \texttt{Succ}, x.p :: \texttt{Nat}) : [y.\ y :: \texttt{Nat}] := x.p
\end{array}$$

2 An Implementation

We provide an implementation of

- the constraint entailment,
- the operational semantics,
- the type assignment and
- the type checking excepting the completeness- and unique check for programs.

The implementation of the operational semantics is straightforward and resembles the rules as seen in Section 1.3.

The implementation of type assignment infers a suitable type for a given expression (based on the rules of Section 1.4). Checking if a given expression has a suspected type is done via first inferring a type for the expression and then checking if the inferred type is a subtype of the suspected type (rule T-Sub).

The implementation of type checking (well-formedness, Section 1.5) is straightforward, with the limitation that we skip the completeness- and unique check for programs.

Our approach for the implementation of the constraint entailment rules from Section 1.2 is to use a SMT solver to solve the constraint system, s.t. for each entailment to solve we make a query to the solver. This requires us to provide an encoding of the rules into (first-order) logic to feed as input into the solver.

The code can be found in the dep-classes-smt github repo.

3 Towards an SMT encoding

Since we want to use a SMT solver for solving constraint entailments we need to encode the calculus into (first-order) logic. Creating this encoding is an iterative process for which we sketch the individual steps we did.

Our first approach was to resemble the syntactic-reasoning of the sequent calculus (1.2) with the encoding. For this we defined ADTs for the three constraint types as well as for paths and lists and we used strings for representing variables, fields and classes. This enabled us to encode the calculus rules in a highly computational form, which made extensive use of recursive definitions ranging over the ADTs. This performed badly. We were able to successfully show simple valid entailments, but the solver gave up and returned `unknown` on more complex valid entailments. Also the solver was not able to reject invalid entailments.

As it turned out, relying on the SMT solver to perform multiple recursive computations is problematic. Hence we want to use a more semantics based approach of reasoning. So instead of having lists of constraints which where instantiations of an ADT, we want to employ simple boolean reasoning. For this we transform the ADTs of the three constraint types into boolean predicates and keep them unspecified. We also turn classes and fields and variables into enumeration types as they are known from the context of the entailment to check and the program,

$$
\begin{aligned}
&\texttt{Path} ::= \texttt{Variable} \mid \texttt{Path.Field}\\
&p_{\{\!\!\{x \mapsto q\}\!\!\}} := p \ \textbf{match} \ \{\\
&\quad x \Rightarrow (p = x) \ ? \ q \ : \ p\\
&\quad p'.f \Rightarrow p'_{\{\!\!\{x \mapsto q\}\!\!\}}.f\\
&\}\\
&\forall p, c, r, s, x.\\
&\quad p_{\{\!\!\{x \mapsto r\}\!\!\}} :: c \wedge s \equiv r \rightarrow\\
&\qquad p_{\{\!\!\{x \mapsto s\}\!\!\}} :: c
\end{aligned}
$$

Figure 1: Excerpt of Semantics Based Encoding with Computive Substitution

but we keep the ADT for paths as we still need it for computing substitutions. An excerpt of this is shown in Figure 1.

This improved the capability to accept valid entailments, but the solver is still not able to reject invalid entailments.

4 SMT solver limitations

Dealing with quantifiers and recursive definitions is hard for SMT solvers. Since SMT solvers do not use induction, proving any property that requires induction will fail if we want to use an off-the-shelf SMT solver. We might encounter a counterexample during unrolling of the recursive definitions, but if none exists we will get looping behaviour during e-matching.

5 Path Depth Limit Encoding

As mentioned in Section 4, the use of recursive definitions in the encoding has the effect that the SMT solver will in general not be able to find a solution. The recusive definitions that remain in the encoding are the path datatype (infinite domain) and the computation of substitutions. The idea to get rid of these is to set a limit to the path depth, meaning a restricting to witch point paths might be expanded. E.g. if we set the depth limit to one the path `x.f` would be within the limit while `x.f.g` would exceed the limit. With a limit on the maximum path depth in place we can enumerate all possible paths and transform the path datatype from an ADT to an enumeration type.

Until now we relied on the SMT solver to compute substitutions, but the path enumeration type allows us to change how we deal with substitutions. We can now transform the computive substitution function into a boolean predicate, like we did with the constraints. We define the predicate with the signature:

$$\textit{substitute} : \texttt{Path} \times \texttt{Variable} \times \texttt{Path} \times \texttt{Path}$$

For $\textit{substitute}(p, x, q, r)$ we write $p_{\{\!\!\{x \mapsto q\}\!\!\}} = r$, giving us the notion that the predicate should be true if r is the result of the substitution of x with q in p.

Since all existing paths are now known prior to the solver, we can compute each possible substitution externally and define the substitution predicate to be true for exactly these inputs.

The quantified calculus rules need to be updated to respect the new substitution style, e.g. we must check for correct substitutions in the premise of the rule. This transformation is shown in Figure 2.

$$
\begin{aligned}
&\texttt{Path} := \{x, y\}\\
&p_{\{\!\{v \mapsto q\}\!\}} = s :=\\
&\quad (p = \texttt{x} \land v = \texttt{x} \land q = \texttt{y} \land s = \texttt{y}) \lor\\
&\quad (p = \texttt{x} \land v = \texttt{y} \land q = \texttt{y} \land s = \texttt{x}) \lor\\
&\quad (p = \texttt{y} \land v = \texttt{x} \land q = \texttt{x} \land s = \texttt{x}) \lor\\
&\quad (p = \texttt{y} \land v = \texttt{y} \land q = \texttt{x} \land s = \texttt{y}) \lor\\
&\quad \ldots\\
&\forall p, c, v, r, s, a, b.\\
&\quad s \equiv r \land p_{\{\!\{v \mapsto r\}\!\}} = a \land\\
&\quad a :: c \land p_{\{\!\{v \mapsto s\}\!\}} = b\\
&\qquad \to b :: c
\end{aligned}
$$

Figure 2: Encoding Excerpt with Enumerated Paths

5.1 Rule C-Prog

With the path depth limit in place, we need to enumerate the C-Prog rule. This is because the rule instantiates a declaration of the program and this declaration introduces a variable that gets substituted in the rule. We cannot defer the substitution check to the SMT solver (like we did in the other rules), since the variable introduced by the declaration is only used as an intermediate value. This variable should therefore not be present outside of this rule, but having the substitution check in the solver would require exactly this.

This requires us to compute the substitution (think of it as intantiating the declaration with a concrete path) prior to the solver, which is possible since all valid paths are known a priori.

5.2 Decidability

The encoding is decidable since we can finitely enumerate the quantifiers, as all quantified variables range over a finite domain. This is because all declared datatypes are enumeration types with a finite amount of constructors and all quantified variables range over one of the declared enumeration types.

5.3 Example Encoding

Assume the natural numbers program from Section 1.6. We want to encode

$$x.\mathbf{cls} \equiv \mathtt{Succ}, x.p.\mathbf{cls} \equiv \mathtt{Zero}, x \equiv y \vdash y :: \mathtt{Nat}$$

using a depth limit of 1.

$$\begin{aligned}
&\mathtt{Variable} := \{\mathtt{x}, \mathtt{y}\} && && (1)\\
&\mathtt{Class} := \{\mathtt{Zero}, \mathtt{Succ}, \mathtt{Nat}\} && && (2)\\
&\mathtt{Path} := \{\mathtt{x}, \mathtt{x.p}, \mathtt{y}, \mathtt{y.p}\} && && (3)\\
&p_{\{\!\{v \mapsto q\}\!\}} = s := && && (4)\\
&\quad (p = \mathtt{x} \land v = \mathtt{x} \land q = \mathtt{x} \land s = \mathtt{x}) \lor && && (5)\\
&\quad (p = \mathtt{x} \land v = \mathtt{x} \land q = \mathtt{x.p} \land s = \mathtt{x.p}) \lor && && (6)\\
&\quad (p = \mathtt{x} \land v = \mathtt{x} \land q = \mathtt{y} \land s = \mathtt{y}) \lor && && (7)\\
&\quad (p = \mathtt{x} \land v = \mathtt{x} \land q = \mathtt{y.p} \land s = \mathtt{y.p}) \lor && && (8)\\
&\quad (p = \mathtt{x.p} \land v = \mathtt{x} \land q = \mathtt{x} \land s = \mathtt{x.p}) \lor && && (9)\\
&\quad (p = \mathtt{x.p} \land v = \mathtt{x} \land q = \mathtt{y} \land s = \mathtt{y.p}) \lor && && (10)\\
&\quad \ldots && && (11)\\
&\forall p.\ p \equiv p && \text{(C-Refl)} && (12)\\
&\forall p, c.\ p.\mathbf{cls} \equiv c \to p :: c && \text{(C-Class)} && (13)\\
&\forall p, q, v, r, s, a, b, c, d. && \text{(C-Subst)} && (14)\\
&\quad s \equiv r \land p_{\{\!\{v \mapsto r\}\!\}} = a \land q_{\{\!\{v \mapsto r\}\!\}} = b \land && && (15)\\
&\quad a \equiv b \land p_{\{\!\{v \mapsto s\}\!\}} = c \land q_{\{\!\{v \mapsto s\}\!\}} = d && && (16)\\
&\qquad \to c \equiv d && && (17)\\
&\forall p, c, v, r, s, a, b. && \text{(C-Subst)} && (18)\\
&\quad s \equiv r \land p_{\{\!\{v \mapsto r\}\!\}} = a \land && && (19)\\
&\quad a :: c \land p_{\{\!\{v \mapsto s\}\!\}} = b && && (20)\\
&\qquad \to b :: c && && (21)\\
&\forall p, c, v, r, s, a, b. && \text{(C-Subst)} && (22)\\
&\quad s \equiv r \land p_{\{\!\{v \mapsto r\}\!\}} = a \land && && (23)\\
&\quad a.\mathbf{cls} \equiv c \land p_{\{\!\{v \mapsto s\}\!\}} = b && && (24)\\
&\qquad \to b.\mathbf{cls} \equiv c && && (25)\\
&\mathtt{x} :: \mathtt{Zero} \to \mathtt{x} :: \mathtt{Nat} && \text{(C-Prog)} && (26)\\
&\mathtt{x.p} :: \mathtt{Zero} \to \mathtt{x.p} :: \mathtt{Nat} && \text{(C-Prog)} && (27)\\
&\mathtt{x} :: \mathtt{Succ} \land \mathtt{x.p} :: \mathtt{Nat} \to \mathtt{x} :: \mathtt{Nat} && \text{(C-Prog)} && (28)\\
&\mathtt{y} :: \mathtt{Zero} \to \mathtt{y} :: \mathtt{Nat} && \text{(C-Prog)} && (29)\\
&\mathtt{y.p} :: \mathtt{Zero} \to \mathtt{y.p} :: \mathtt{Nat} && \text{(C-Prog)} && (30)\\
&\mathtt{y} :: \mathtt{Succ} \land \mathtt{y.p} :: \mathtt{Nat} \to \mathtt{y} :: \mathtt{Nat} && \text{(C-Prog)} && (31)\\
&\neg(\mathtt{x}.\mathbf{cls} \equiv \mathtt{Succ} \land \mathtt{x.p}.\mathbf{cls} \equiv \mathtt{Zero} \land \mathtt{x} \equiv \mathtt{y} \to \mathtt{y} :: \mathtt{Nat}) && && (32)
\end{aligned}$$

6 Grounding the Path Depth Limit Encoding

With the Path Depth Limit we have a (assumed) decidable encoding, but we are still using quantifiers. An idea is to ground the encoding. This could result in a better solving performance, as having quantifiers in the encoding and therefore needing to utilize quantifier instantiation in the solver adds complexity. Instead, using a quantifier free encoding might be beneficial. It might also enable us to employ additional optimizations.

As discussed in Section 5.2 it is possible to finitely enumerate all quantified variables. So our first approach to ground the encoding is to do exactly this and assert each single quantifier instantiation in a big conjunction.

Example Given the rule for substitutions over instance-of constraints (see Figure 2), we instantiate the rule for each combination of paths p, r, s, a, b, variables v and classes c to obtain a ground formula. Assume $p = r = s = b = \mathtt{x}$, $v = a = \mathtt{y}$ for some class $\mathtt{C}$, we obtain the instantiation:

$$\begin{aligned}&\mathtt{x} \equiv \mathtt{x} \land \mathtt{x}_{\{\!\{\mathtt{y}\mapsto\mathtt{x}\}\!\}} = \mathtt{y} \ \land \mathtt{y} :: \mathtt{C} \land \mathtt{x}_{\{\!\{\mathtt{y}\mapsto\mathtt{x}\}\!\}} = \mathtt{x}\\ &\quad\rightarrow \mathtt{x} :: \mathtt{C}\end{aligned}$$

6.1 Substitution in the Ground Encoding

If we inspect the previous example, we can make the following observations:

1. Some of the substitution checks will be statically known to be false and
2. all substitution checks can be statically decided.

The previous example is an instantiation of the rule:

$$\begin{aligned}&\forall p, c, v, r, s, a, b.\\ &\quad s \equiv r \land p_{\{\!\{v\mapsto r\}\!\}} = a \ \land a :: c \land p_{\{\!\{v\mapsto s\}\!\}} = b\\ &\qquad\rightarrow b :: c\end{aligned} \tag{C-Subst}$$

The substitution check marked in red is statically/externally known to be false and the one marked in blue is known to be true.

$$\begin{aligned}&\mathtt{x} \equiv \mathtt{x} \land \mathtt{x}_{\{\!\{\mathtt{y}\mapsto\mathtt{x}\}\!\}} = \mathtt{y} \ \land \mathtt{y} :: \mathtt{C} \land \mathtt{x}_{\{\!\{\mathtt{y}\mapsto\mathtt{x}\}\!\}} = \mathtt{x}\\ &\quad\rightarrow \mathtt{x} :: \mathtt{C}\end{aligned}$$

From this we can conclude that this instantiation will never be used in prvoing the property $\mathtt{x} :: \mathtt{C}$, as the premise of the implication can never be fulfilled.

By inspecting another instantiation of the same rule, e.g. with $p = \mathtt{x}, v = \mathtt{x}, r = \mathtt{y}, a = \mathtt{y}, s = \mathtt{x}, b = \mathtt{x}, c = \mathtt{C}$ we obtain the formula:

$$\begin{aligned}&\mathtt{x} \equiv \mathtt{y} \land \mathtt{x}_{\{\!\{\mathtt{x}\mapsto\mathtt{y}\}\!\}} = \mathtt{y} \ \land \mathtt{y} :: \mathtt{C} \land \mathtt{x}_{\{\!\{\mathtt{x}\mapsto\mathtt{x}\}\!\}} = \mathtt{x}\\ &\quad\rightarrow \mathtt{x} :: \mathtt{C}\end{aligned}$$

In this instantiation both substitution checks are known to be true and the rule might be usable in proving that $\mathtt{x} :: \mathtt{C}$.

With these observations we can modify the ground encoding as follows:

1. We do not have to include formulae in the encoding that are not helping in showing a property. E.g. where the premise of an implication is known to be false.
2. Since we can statically decide all substitution checks, we do not have to encode the substitution predicate. This is in synergy with the previous point, as removing the checks in the implications gets possible if we only add those formulae where the checks are guaranteed to be true.

These changes reduce the complexity for the SMT solver as well as for the generation of the query to be sent to the solver (preprocessing). The SMT solver complexity is reduced through two steps. (1) The removal of the substitution predicate removed the need for the solver to check and (2) the total amount of possibilities to check is lowered, since the rules sent to the solver only include those cases where the substitution check would have been true.

The reduction in complexity in the preprocessing is mainly due to the fact that we do not have to generate the definition for the substitution predicate anymore, which involved iterating over the cross product of the input parameters.

Example If we consider the two shown example instantiations, we would only take the second one

$$\begin{aligned} &\mathtt{x} \equiv \mathtt{y} \wedge \mathtt{x}_{\{\mathtt{x} \mapsto \mathtt{y}\}} = \mathtt{y} \ \wedge \mathtt{y} :: \mathtt{C} \wedge \mathtt{x}_{\{\mathtt{x} \mapsto \mathtt{x}\}} = \mathtt{x} \\ &\quad \to \mathtt{x} :: \mathtt{C} \end{aligned}$$

which with the mentioned modifications turns into the formula

$$\mathtt{x} \equiv \mathtt{y} \wedge \mathtt{y} :: \mathtt{C} \to \mathtt{x} :: \mathtt{C}$$

6.2 Decidability

The encoding is decidable, as all declared datatypes have a finite domain and the encoding is quantifier free.

6.3 Example Encoding

Assume the program of natural numbers given in Section 1.6. We want to encode

$$x.\mathbf{cls} \equiv \texttt{Succ}, x.p.\mathbf{cls} \equiv \texttt{Zero}, x \equiv y \vdash y :: \texttt{Nat}$$

using a path depth limit of 1.

$$\texttt{Variable} := \{\texttt{x}, \texttt{y}\} \tag{1}$$
$$\texttt{Class} := \{\texttt{Zero}, \texttt{Succ}, \texttt{Nat}\} \tag{2}$$
$$\texttt{Path} := \{\texttt{x}, \texttt{x.p}, \texttt{y}, \texttt{y.p}\} \tag{3}$$
$$\texttt{x} \equiv \texttt{x} \wedge \texttt{x.p} \equiv \texttt{x.p} \wedge \texttt{y} \equiv \texttt{y} \wedge \texttt{y.p} \equiv \texttt{y.p} \qquad \text{(C-Refl)} \tag{4}$$
$$\texttt{x}.\mathbf{cls} \equiv \texttt{Zero} \to \texttt{x} :: \texttt{Zero} \qquad \text{(C-Class)} \tag{5}$$
$$\texttt{x.p}.\mathbf{cls} \equiv \texttt{Zero} \to \texttt{x.p} :: \texttt{Zero} \qquad \text{(C-Class)} \tag{6}$$
$$\ldots \qquad \text{(C-Class)} \tag{7}$$
$$\texttt{x} \equiv \texttt{y} \wedge \texttt{y} \equiv \texttt{y} \to \texttt{y} \equiv \texttt{x} \qquad \text{(C-Subst)} \tag{8}$$
$$\texttt{x} \equiv \texttt{y} \wedge \texttt{y} :: Nat \to \texttt{x} :: \texttt{Nat} \qquad \text{(C-Subst)} \tag{9}$$
$$\texttt{x} \equiv \texttt{y} \wedge \texttt{y}.\mathbf{cls} \equiv Nat \to \texttt{x}.\mathbf{cls} \equiv \texttt{Nat} \qquad \text{(C-Subst)} \tag{10}$$
$$\ldots \qquad \text{(C-Subst)} \tag{11}$$
$$\texttt{x} :: \texttt{Zero} \to \texttt{x} :: \texttt{Nat} \qquad \text{(C-Prog)} \tag{12}$$
$$\texttt{x.p} :: \texttt{Zero} \to \texttt{x.p} :: \texttt{Nat} \qquad \text{(C-Prog)} \tag{13}$$
$$\texttt{x} :: \texttt{Succ} \wedge \texttt{x.p} :: \texttt{Nat} \to \texttt{x} :: \texttt{Nat} \qquad \text{(C-Prog)} \tag{14}$$
$$\texttt{y} :: \texttt{Zero} \to \texttt{y} :: \texttt{Nat} \qquad \text{(C-Prog)} \tag{15}$$
$$\texttt{y.p} :: \texttt{Zero} \to \texttt{y.p} :: \texttt{Nat} \qquad \text{(C-Prog)} \tag{16}$$
$$\texttt{y} :: \texttt{Succ} \wedge \texttt{y.p} :: \texttt{Nat} \to \texttt{y} :: \texttt{Nat} \qquad \text{(C-Prog)} \tag{17}$$
$$\ldots \tag{18}$$
$$\neg(\texttt{x}.\mathbf{cls} \equiv \texttt{Succ} \wedge \texttt{x.p}.\mathbf{cls} \equiv \texttt{Zero} \wedge \texttt{x} \equiv \texttt{y} \to y :: \texttt{Nat}) \tag{19}$$

7 Determining a suitable depth limit

We have introduced an encoding that relies on a parameter that limits the maximum path depth. How do we set this limit and is it possible to set it such that the solver finds a solution if one would exists in the sequent calculus.

Lemma 1 (5.5.16)**.** *If* $\mathsf{wf}P$ *then* $\overline{a} \vdash a$ *iff* $\overline{a} \vdash_A a$.

Lemma 2 (5.5.1)**.** $\overline{a} \vdash_A a$ *is decidable.*

For this, we rely to the algorithmic system and the equivalency between the algorithmic- and the declarative system (Lemma 1) and the decidability of the algorithmic system (Lemma 2).

The proof of Lemma 2 defines a set of paths $S''_{\overline{a};a}$ and proves that all judgements in the derivation contain only paths from this set.

We use this to set the depth limit parameter for the encoding of the given entailment to solve. For this we determine the path $p \in S''_{\overline{a};a}$ s.t.

$$\forall p' \in S''_{\overline{a};a}.\ \mathit{depth}(p) \geq \mathit{depth}(p')$$

and set the depth limit used for the encoding to be precisely $\mathit{depth}(p)$.

8 Algorithmic Symmetry

As emphasized in Section 7 we rely on the equivalency between the declarative and the algorithmic system to set our depth limit for path enumeration as well as on the decidability of the declarative system to even have such a limit in place.

Lemma 3 (5.5.15)**.** *If* $\mathsf{wf}P$ *and* $\overline{a} \vdash a$ *then* $\overline{a} \vdash_A a$.

Theorem 1 (5.5.1)**.** *If* $\mathsf{wf}P$ *then derivation of* $\overline{a} \vdash a$ *is decidable.*

The entailment $a \equiv b \vdash b \equiv a$ is a counterexample to Lemma 3 which describes one direction of Lemma 1 and Theorem 1 as it relies on Lemma 1.

Counterexample Choose any well-formed program.

The entailment $a \equiv b \vdash b \equiv a$ has a derivation in the declarative system.

$$\dfrac{\dfrac{\dfrac{}{\cdot \vdash b \equiv b}\;\text{C-Refl}}{a \equiv b \vdash b \equiv a_{\{\!\{a \mapsto b\}\!\}}}\;\text{C-Weak} \qquad \dfrac{}{a \equiv b \vdash a \equiv b}\;\text{C-Ident}}{a \equiv b \vdash b \equiv a_{\{\!\{a \mapsto a\}\!\}}}\;\text{C-Subst}$$

In the algorithmic system, only rule CA-Subst3 applies. We have two possible ways to approach this sequent.

$$\dfrac{\dfrac{}{a \equiv b \vdash_A a \equiv a}\;\text{CA-Refl} \qquad a \sqsubset a \equiv b \qquad \dfrac{\dots}{a \equiv b \vdash_A b \equiv a}}{a \equiv b \vdash_A b \equiv a}\;\text{CA-Subst3}$$

$$\dfrac{\dfrac{\dots}{a \equiv b \vdash_A b \equiv a} \qquad b \sqsubset a \equiv b \qquad \dfrac{}{a \equiv b \vdash_A b \equiv b}\;\text{CA-Refl}}{a \equiv b \vdash_A b \equiv a}\;\text{CA-Subst3}$$

$$\textsc{CA-Refl}\quad \overline{a} \vdash_A p \equiv p \qquad \textsc{CA-Ident}\quad \overline{a} \vdash_A a_i$$

$$\textsc{CA-Class}\quad \frac{\overline{a} \vdash_A p.\mathbf{cls} \equiv C}{\overline{a} \vdash_A p :: C} \qquad \textsc{CA-Prog}\quad \frac{\overline{a} \vdash_A \overline{b}_{\{\!\{x \mapsto p\}\!\}} \quad p \sqsubset \overline{a} \quad (\forall x.\ \overline{b} \Rightarrow x :: C) \in P}{\overline{a} \vdash_A p :: C}$$

$$\textsc{CA-Subst1}\quad \frac{p \sqsubset \overline{a} \quad \overline{a} \vdash_A p.\mathbf{cls} \equiv C \quad \overline{a} \vdash_A p \equiv p'}{\overline{a} \vdash_A p'.\mathbf{cls} \equiv C} \qquad \textsc{CA-Subst2}\quad \frac{p \sqsubset \overline{a} \quad \overline{a} \vdash_A p :: C \quad \overline{a} \vdash_A p \equiv p'}{\overline{a} \vdash_A p' :: C}$$

$$\textsc{CA-Subst3}\quad \frac{p \sqsubset \overline{a} \quad \overline{a} \vdash_A p \equiv p'' \quad \overline{a} \vdash_A p' \equiv p}{\overline{a} \vdash_A p' \equiv p''} \qquad \textsc{CA-Subst4}\quad \frac{\overline{a} \vdash_A p \equiv p'}{\overline{a} \vdash_A p.f \equiv p'.f}$$

Figure 3: Algorithmic Rules

We can see that we reproduce the same proof goal in one of the branches of both possibilities and we cannot close those branches.

8.1 Symmetry Fix

We can update rule CA-Subst3 to allow the entailment used as a counterexample to Lemma 3 to have a derivation.

There are two feasible ways to update the rule:

1. $$\frac{\overline{a} \vdash_A p \equiv p'' \qquad p \sqsubset \overline{a} \qquad \overline{a} \vdash_A p \equiv p'}{\overline{a} \vdash_A p' \equiv p''}\ \text{CA-Subst3Fix1}$$

2. $$\frac{\overline{a} \vdash_A p'' \equiv p \qquad p \sqsubset \overline{a} \qquad \overline{a} \vdash_A p' \equiv p}{\overline{a} \vdash_A p' \equiv p''}\ \text{CA-Subst3Fix2}$$

8.1.1 Fix 1 derivation

$$\frac{\dfrac{}{a \equiv b \vdash_A a \equiv a}\ \text{CA-Refl} \qquad a \sqsubset a \equiv b \qquad \dfrac{}{a \equiv b \vdash_A a \equiv b}\ \text{CA-Ident}}{a \equiv b \vdash_A b \equiv a}\ \text{CA-Subst3Fix1}$$

8.1.2 Fix 2 derivation

$$\frac{\dfrac{}{a \equiv b \vdash_A a \equiv b}\ \text{CA-Ident} \qquad b \sqsubset a \equiv b \qquad \dfrac{}{a \equiv b \vdash_A b \equiv b}\ \text{CA-Refl}}{a \equiv b \vdash_A b \equiv a}\ \text{CA-Subst3Fix2}$$

8.1.3 Fix Implications

Deploying the update to rule CA-Subst3 might have implications on the termination of the algorithmic system. We could e.g. repeatedly apply rule CA-Subst3 to symmetrically switch a path equivalence.

This can be resolved by having the limitaion that we may apply a rule using the same parameters at most once per branch.

This seems sensible, as we (most likely) already need this for termination. Otherwise we would also be allowed to satisfy $p \sqsubset \overline{a}$ for the same p.

9 Object Construction

With the rules for type assignment as seen in Section 1.4 it is possible to construct objects with more fields than anticipated. This is because the constructor declaration defines a set of constraints that need to be fulfilled in order to create a new object, but it is possible to ”oversatisfy” those constraints.

Example

$$\dfrac{\begin{array}{cc} x :: \texttt{Zero} \vdash x : [y.p.\ y.p \equiv x] & \overline{b} = y.\mathbf{cls} \equiv \texttt{Zero}, y.p \equiv x \\ \texttt{Zero}(y.\ \epsilon) \in P & x :: \texttt{Zero}, \overline{b} \vdash \epsilon \end{array}}{x :: \texttt{Zero} \vdash \mathbf{new}\ \texttt{Zero}(p \equiv x) : [y.\ y.\mathbf{cls} \equiv \texttt{Zero}, y.p \equiv x]}\ \text{T-New}$$

Since anything entails ϵ, the check for $x :: \texttt{Zero}, y.\mathbf{cls} \equiv \texttt{Zero}, y.p \equiv x \vdash \epsilon$ succeeds and we can successfully assign a type for $\mathbf{new}\ \texttt{Zero}(p \equiv x)$, albeit the constructor for `Zero` doesn't constraint any fields.

This is because of the discrepancy between the syntax of object construction $\mathbf{new}\ C(\overline{f} \equiv \overline{e})$ and the syntax of constructor declaration $C(x.\ \overline{a})$ and the way they come together in rule T-New. In object creation we have a concrete set of fields $\overline{f}$ we want to assign values to, while in constructor declaration we only have a set of constraints $\overline{a}$ that describe the class including it's fields.

In rule T-New we check with $\overline{c}, \overline{b} \vdash \overline{b'}$ that we supplied enough fields to satisfy the constraints of the class, but we do not check the other direction. Namely that we provide fields that are unknown to the constraints of the constructor.

9.1 Update to T-New

We add an additional check to rule T-New that checks if the fields $\overline{f}$ are constrained from the constructor.

$$\text{T-New}\quad \dfrac{\begin{array}{c} \forall i.\ \overline{c} \vdash e_i : [x_i.\ \overline{a_i}] \qquad \overline{b} = (x.\mathbf{cls} \equiv C), \cup_i \overline{a_i}\{\!\!\{x_i \mapsto x.f_i\}\!\!\} \\ C(x.\ \overline{b'}) \in P \qquad \overline{c}, \overline{b} \vdash \overline{b'} \qquad \forall i.\ f_i \in \mathsf{FIELDS}(b') \end{array}}{\overline{c} \vdash \mathbf{new}\ C(\overline{f} \equiv \overline{e}) : [x.\ \overline{b}]}$$

10 Refinement Types

Can we relate the DCC system to refinement types and how does DCCs constraint language compare in terms of expressiveness to what we expect from refinement types?

Usually refinement types are depicted as $\{x : \tau \mid \varphi\}$, where τ is a type and φ is a formula over x.

10.1 Semantics of Refinement Types

Semantically, such a type is meant to refine the domain of the type τ with φ. E.g. the type $\{n : \texttt{Int} \mid n > 0\}$ refines the domain of all integers to only the strictly positive integers.

$$[\![\{x : \tau \mid \varphi\}]\!] = \{v | v \in [\![\tau]\!] \wedge \varphi_{\{\![x \mapsto v]\!\}}\}$$

10.2 Connection to DCC

We can relate multiple constructs from DCC to refinement types, or at least state that those behave in a similar way to the semantics refinements expressable with refinement types.

10.3 Constructor Declaration

We can think of DCCs constructor declaration as some sort of refinement of the objects of a class. E.g. the declaration $C(x.\ \overline{a})$ to semantically express something similar to a potential refinement type $\{x : C \mid \overline{a}\}$. Refining objects of class C to adhere to $\overline{a}$.

10.4 What is an Object in DCC?

Objects usually consist of structure (fields) and behaviour (methods). In DCC objects only consists of structure (fields), while the behaviour (methods) is declared outside the scope of classes and their objects. More specific, their are only constructor declarations and method declarations at the "top level" of the syntax instead of class declarations that encapsulate/incorporate both. In DCC an object $\langle C; \overline{f \equiv x} \rangle$ on the heap consists of 1) a class C from which the object was constructed from and 2) a list of pairs $f_i \equiv x_i$ of the fields of the object and their corresponding memory location on the heap.

10.5 Types

In DCC a type $[x.\ \overline{a}]$ consist of 1) an identifier x and 2) a set of constraints $\overline{a}$ over x. Semantically, such a type describes all objects that fulfill constraints $\overline{a}$. This can be seen as a refinement type over objects $\{x : \mathsf{Object} \mid \overline{a}\}$. E.g. $\{n : \texttt{Int} \mid n > 0\}$ would translate to $[n.\ n :: \texttt{Int}, n > \mathit{0}]$, ignoring the fact that there is a non-structural constraint $n > 0$.

Since DCCs types are only concerned with objects and their structure we omit the Object type in the type syntax and instead of a generic formulae φ we have a set of constraints $\overline{a}$. This is possible, since the class an object belongs to (as well as inheritance) is handled through the constraint system. Semantically, a DCC type behaves similar to a refinement type as statet previously and describes objects that conform to the constraints of the type, more formally:

$$[\![[x.\ \overline{a}]]\!] = \{o | o \in \mathsf{Object} \wedge OC(x, o) \vdash \overline{a}\}$$

10.6 DCC refinement limitations

Refinement types usually use formulae for their refinement (with varying restrictions). In DCC we the refinements are limited to a set of (structural) constraints and all of those need to be fulfilled simultaneously. This is equivalent to a big conjunction in the language of predicate logic.

Interestingly, this is exactly how the translation in the SMT encoding works. E.g. a set of constraints (a_1, a_2, a_3) gets translated to $(\mathsf{SMT}(a_1) \land \mathsf{SMT}(a_2) \land \mathsf{SMT}(a_3))$ or more generally $\overline{a}$ translates to $\bigwedge_i \mathsf{SMT}(a_i)$. The empty set of constraints ϵ translates to *true*.

10.7 future work

A possible direction for future work is to extend the expressiveness of the refinements (constraint system). There are two paths to extend upon:

1. transform the strict conjunctive form of the current constraints to support a wider range of operations through support for negation etc.
2. add new kinds of constraints that are not neccecarily structural.