STLC + μ

Safety and Semantic Approach

Cleo Gerards & Dick Blankvoort

Radboud University Nijmegen

27 November 2023

Contents

Introduction

Introduction

Introduction

PCC = ‘Proof-carrying code’
Ensuring a trusted program does no harm

Introduction

PCC = 'Proof-carrying code'
Ensuring a trusted program does no harm

We introduce new type semantics to reduce the complexity of proofs

Semantic approach

Semantic proof consists of the following steps:

1. Assign meaning to type judgments
2. Proof that if a type judgment is true, then the typed machine state is safe
3. Proof that type inference rules are safe

Semantic approach

Semantic proof consists of the following steps:

1. Assign meaning to type judgments
2. Proof that if a type judgment is true, then the typed machine state is safe
3. Proof that type inference rules are safe

We then know that the derivable type judgments are true, and thus the typable machine states are safe.

Semantic approach

Avoid formalizing syntactic type expressions $\Longrightarrow$ Formalize a type as a set of semantic values

Semantic approach

Avoid formalizing syntactic type expressions $\Longrightarrow$ Formalize a type as a set of semantic values

Definition

We define the operator $\rightarrow$ as a function taking two sets as arguments and returning a set

Semantic approach

Thus, replace the inference rule:

$$\frac{\Gamma \vdash f : \alpha \to \beta, \quad \Gamma \vdash e : \alpha}{\Gamma \vdash (f \ e) : \beta}$$

for the semantic lemma:

$$\frac{\Gamma \vDash f : \alpha \to \beta, \quad \Gamma \vDash e : \alpha}{\Gamma \vDash (f \ e) : \beta}$$

Indexed types for the lambda calculus

Indexed types for the lambda calculus

Recursive types with Cartesian products and constant 0

We define the syntax of lambda terms with products and **0** by the following grammar:

$$e ::= x \mid \mathbf{0} \mid \langle e_1, e_2 \rangle \mid \pi_1(e) \mid \pi_2(e) \mid \lambda x.e \mid (e_1 e_2)$$

Recursive types with Cartesian products and constant 0

We define the syntax of lambda terms with products and **0** by the following grammar:

$$e ::= x \mid \mathbf{0} \mid \langle e_1, e_2 \rangle \mid \pi_1(e) \mid \pi_2(e) \mid \lambda x.e \mid (e_1 e_2)$$

Definition

A term v is a **value** if it is $\mathbf{0}$, a closed term of the form $\lambda x.e$, or a pair $\langle v_1, v_2 \rangle$ of values.

Definitions

Definition

We write $e \mapsto^j e'$ to mean that there exists a chain of j steps of the form $e \mapsto e_1 \mapsto \ldots \mapsto e_j = e'$.
We write e $\mapsto^* e'$ if $e \mapsto^j e'$ for some $j \geq 0$.

Definitions

Definition

We write $e \mapsto^j e'$ to mean that there exists a chain of j steps of the form $e \mapsto e_1 \mapsto \ldots \mapsto e_j = e'$.
We write e $\mapsto^* e'$ if $e \mapsto^j e'$ for some $j \geq 0$.

Definition

A term is **irreducible** if it has no successor in the step relation.

This means that irred(e) if e is a value or e is a "stuck" expression, e.g. $\pi_1(\lambda x.e')$.

Definition of safe

Definition

A term e is **safe for k steps** if for any reduction $e \mapsto^j e'$ of $j < k$ steps, one of the following holds:

- e' is a value
- $e' \mapsto e''$, for some e''

Note: any term is safe for 0 steps.

Definition of safe

Definition

A term e is **safe for** k **steps** if for any reduction $e \mapsto^j e'$ of $j < k$ steps, one of the following holds:

- e' is a value
- $e' \mapsto e''$, for some e''

Note: any term is safe for 0 steps.

Definition

A term e is called **safe** if it is safe for all $k \geq 0$.

Types as sets

Definition

A **type** is a set τ of pairs of the form $\langle k, v\rangle$ where:

- k is a nonnegative integer
- if $\langle k, v\rangle \in \tau$ and $0 \leq j \leq k$, then $\langle j, v\rangle \in \tau$

Types as sets

Definition

A **type** is a set τ of pairs of the form $\langle k, v\rangle$ where:

- k is a nonnegative integer
- if $\langle k, v\rangle \in \tau$ and $0 \leq j \leq k$, then $\langle j, v\rangle \in \tau$

Definition

For any closed expression e and type τ we write $e :_k \tau$ if whenever $e \mapsto^j v$ for $j < k$ and v irreducible, then $\langle k - j, v\rangle \in \tau$.

Or in other words:

$$e :_k \tau \equiv \forall j\ \forall v.\ 0 \leq j \leq k\ \wedge\ e \mapsto^j v\ \wedge\ \mathrm{irred}(v)\ \Rightarrow\ \langle k - j, v\rangle \in \tau$$

Observations

We can observe the following:

- if $e :_k \tau$ and $0 \leq j \leq k$, then $e :_j \tau$
- If v is a value and $k > 0$, then the statements $v :_k \tau$ and $\langle k, v \rangle \in \tau$ are equivalent.

μ operator

Let μ be a function where:

- the input is a set functional F, so a function from sets to sets
- the output is a set that is a fixed point of F

μ operator

Let μ be a function where:

- the input is a set functional F, so a function from sets to sets
- the output is a set that is a fixed point of F

μ allows us to define recursive types:

$$\begin{gathered}
\bot \equiv \{\} \\
\top \equiv \{\langle k, v\rangle \mid k \geq 0\} \\
\textbf{int} \equiv \{\langle k, \mathbf{0}\rangle \mid k \geq 0\} \\
\tau_1 \times \tau_2 \equiv \{\langle k, (v_1, v_2)\rangle \mid \forall j < k. \langle j, v_1\rangle \in \tau_1 \wedge \langle j, v_2\rangle \in \tau_2\} \\
\sigma \to \tau \equiv \{\langle k, \lambda x.e\rangle \mid \forall j < k \; \forall v. \langle j, v\rangle \in \sigma \Rightarrow e[v/x] :_j \tau\} \\
\mu F \equiv \{\langle k, v\rangle \mid \langle k, v\rangle \in F^{k+1}(\bot)\}
\end{gathered}$$

Environments

Definition

A **type environment** is a mapping from lambda calculus variables to types.
A **value environment** is a mapping from lambda calculus variables to values.

Environments

For any type environment Γ and value environment σ we write $\sigma :_k \Gamma$ if for all variables $x \in \mathrm{dom}(\Gamma)$ we have $\sigma(x) :_k \Gamma(x)$.

Environments

For any type environment Γ and value environment σ we write $\sigma :_k \Gamma$ if for all variables $x \in \mathrm{dom}(\Gamma)$ we have $\sigma(x) :_k \Gamma(x)$.

We write $\Gamma \vDash_k e : \alpha$ to say that every free variable of e is mapped by Γ and $\forall \sigma . \sigma :_k \Gamma \Rightarrow \sigma(e) :_k \alpha$. Here, $\sigma(e)$ is the result of replacing the free variables in e with their values under σ.

We write $\Gamma \vDash e : \alpha$ if for all $k \geq 0$, we have $\Gamma \vDash_k e : \alpha$.

We write $\vDash e : \alpha$, to mean $\Gamma_0 \vDash e : \alpha$ for the empty environment Γ_0

First lemma!

Lemma

If $\vDash e : \alpha$, *then* e *is safe.*

Properties of the typing lemmas

Properties of the typing lemmas

Function types

For function types we have that the following properties should hold:

1. Types are closed under function types ($\rightarrow$).

Function types

For function types we have that the following properties should hold:

1. Types are closed under function types ($\rightarrow$).
   - If α and β are types then $\alpha \rightarrow \beta$ is as well.

Function types

For function types we have that the following properties should hold:

1. Types are closed under function types ($\rightarrow$).
   - If α and β are types then $\alpha \rightarrow \beta$ is as well.
2. Function types are well-behaved (semantic rules are satisfied).

Function types

For function types we have that the following properties should hold:

1. Types are closed under function types ($\rightarrow$).
   - If α and β are types then $\alpha \rightarrow \beta$ is as well.
2. Function types are well-behaved (semantic rules are satisfied).
   - Application: under a set context, the existence of a term $e_1 : \alpha \rightarrow \beta$ and a term $e_2 : \alpha$ implies the existence of a term $(e_1 e_2) : \beta$.

Function types

For function types we have that the following properties should hold:

1. Types are closed under function types ($\rightarrow$).
   - If α and β are types then $\alpha \rightarrow \beta$ is as well.
2. Function types are well-behaved (semantic rules are satisfied).
   - Application: under a set context, the existence of a term $e_1 : \alpha \rightarrow \beta$ and a term $e_2 : \alpha$ implies the existence of a term $(e_1 e_2) : \beta$.
   - Abstraction: under a set context Γ, $\Gamma[x := \alpha] \vDash e \in \beta$ implies that $\Gamma \vDash \lambda x.e : \alpha \rightarrow \beta$

Product types

For product types the following properties must hold:

1. Types are closed under product types ($\times$).
2. Product types are well-behaved.
    - Combination: the existence of terms $e_1 :_k \alpha$ and $e_2 :_k \alpha$ implies the existence of a term $\langle e_1, e_2 \rangle :_k \alpha \times \beta$.
    - Projection: the existence of a term $e :_k \alpha \times \beta$ implies the existence of projected terms $\pi_1(e) :_k \alpha$ and $\pi_2(e) :_k \beta$.

Well-foundedness

Well-foundedness

k-approximation

k-approximation: the subset of elements of a set τ whose index is less than some number k.

$$\mathrm{approx}(k, \tau) := \{\langle j, v\rangle \mid j < k, \langle j, v\rangle \in \tau\}$$

Types are closed under approximation.

Well-foundedness

A recursive definition is well-founded if safety for k steps can be decided based purely on knowing the safety of any term for $j < k$ steps.

More succinctly: a recursive definition is well-founded if its subterms require fewer steps to determine type-safety.

Well-founded functional

A well-founded functional is a type-transforming function (/type constructor) F for which for any type τ and $k \geq 0$ we have that

$$\mathrm{approx}(k+1, F(\tau)) = \mathrm{approx}(k+1, F(\mathrm{approx}(k, \tau)))$$

Where once again, types are closed under the operation. This definition demands that elements of the domain require strictly less steps to determine type-safety than the elements of the codomain. In other words: unfolding the definition makes determining type-safety simpler.

WF functional properties

We have the following key properties for well-founded functionals:

1. Applying a well-founded functional j or more times to two different types yields identical types up to approximation j.

   Induction on approximation levels.

WF functional properties

We have the following key properties for well-founded functionals:

1. Applying a well-founded functional j or more times to two different types yields identical types up to approximation j.

   Induction on approximation levels.

2. For a well-founded functional F, $\mu(F)$ is a type.

   Recall that μ repeatedly applies its argument to the bottom element.

WF functional properties (cont'd)

3 We have that $\mathrm{approx}(k, \mu(F)) = \mathrm{approx}(k, F^k \bot)$. In words, we can approximate μ in k steps by applying our function k times to the bottom element.

WF functional properties (cont'd)

3 We have that $\mathrm{approx}(k, \mu(F)) = \mathrm{approx}(k, F^k \bot)$. In words, we can approximate μ in k steps by applying our function k times to the bottom element.

4 We have that $\mathrm{approx}(k, \mu(F)) = \mathrm{approx}(k, F(\mu(F)))$. This proves that the type inference lemmas for μ hold for any well-founded functional.

Nonexpansive type constructor

A nonexpansive type constructor (recall: a function from types to types) is a function F such that

$$\mathrm{approx}(k, F(\tau)) = \mathrm{approx}(k, F(\mathrm{approx}(k, \tau)))$$

Nonexpansive type constructor properties

1. Nonexpansive type constructors are closed under composition.

Nonexpansive type constructor properties

1. Nonexpansive type constructors are closed under composition.
2. Composition of a nonexpansive type constructor with a well-founded one results in a well-founded type constructor.

Nonexpansive type constructor properties

1. Nonexpansive type constructors are closed under composition.
2. Composition of a nonexpansive type constructor with a well-founded one results in a well-founded type constructor.
3. Given two nonexpansive type constructors F and G, $\Lambda\alpha.F\alpha \to G\alpha$ and $\Lambda\alpha.F\alpha \times G\alpha$ are well-founded.

Quantifications

It is possible to define existential and universal quantifications for type constructors, in the following manner:

$$\exists F := \bigcup_{\tau \in \mathsf{type}} F\tau \qquad ; \qquad \forall F := \bigcap_{\tau \in \mathsf{type}} F\tau$$

Quantifications

It is possible to define existential and universal quantifications for type constructors, in the following manner:

$$\exists F := \bigcup_{\tau \in \mathsf{type}} F\tau \qquad ; \qquad \forall F := \bigcap_{\tau \in \mathsf{type}} F\tau$$

For these operations, we have five typing rules. Using these rules we can prove, for instance, that the only functions of type $\forall \alpha . \alpha \to \alpha$ are the empty function and the identity function.