From Type Checking by Recursive Descent to Type Checking by Abstract Machine

Ilya Sergey and Dave Clarke

DistriNet & IBBT
Katholieke Universiteit Leuven
{ilya.sergey,dave.clarke}@cs.kuleuven.be

LDTA '11
26 March 2011
Saarbrücken, Germany

From Type Checking by Recursive Descent to Type Checking by Abstract Machine

Ilya Sergey and Dave Clarke

DistriNet & IBBT
Katholieke Universiteit Leuven
{ilya.sergey,dave.clarke}@cs.kuleuven.be

LDTA '11
26 March 2011
Saarbrücken, Germany

This work was carried out while the first author was visiting the BRICS PhD School of Aarhus University in September 2010

What is a program semantics?

What is a program semantics?

It is the meaning
of grammatically correct programs

What is a program semantics?

It is the meaning
of grammatically correct programs

Example of a meaning:

What is a program semantics?

It is the meaning
of grammatically correct programs

Example of a meaning: ***types***

Outline

Outline

A parade of semantics

Different semantics are aimed to answer different questions about programs:

A parade of semantics

Different semantics are aimed to answer different questions about programs:

- *Denotational semantics*: what does a program mean as a mathematical object
  - C. Strachey, D. Scott

A parade of semantics

Different semantics are aimed to answer different questions about programs:

- *Denotational semantics*: what does a program mean as a mathematical object
  - C. Strachey, D. Scott
- *Operational semantics*: how to compute a program on some abstract machine, what is its result
  - G. Plotkin

A parade of semantics

Different semantics are aimed to answer different questions about programs:

- *Denotational semantics*: what does a program mean as a mathematical object
  - C. Strachey, D. Scott
- *Operational semantics*: how to compute a program on some abstract machine, what is its result
  - G. Plotkin
- *Axiomatic semantics*: what are properties of the effect of executing a program
  - R.W.Floyd, C.A.R.Hoare

Denotational semanitcs gives an intuition about “what a program is”, but doesn’t say how to execute it.

Denotational semanitcs gives an intuition about “what a program is”, but doesn’t say how to execute it.

Operational semantics defines how to *execute a program*.

A diversity of operational semantics

- **Big-step (*natural*) semantics**:
  - program evaluation defined inductively on its syntax
  - computes a *fold* over a program's AST
- ***Big-step abstract machine***
  - Execution traces instead of trees
- ***Small-step operational semantics***
  - Each step: decompose-contract-recompose
- ***Reduction semantics***
  - Contexts and contractions
- ***Small-step abstract machine***
  - Examples: CC, SCC, CK, CEK-machines, Krivine's machine, Landin's SECD etc.

A diversity of operational semantics

- **Big-step (*natural*) semantics**:
  - program evaluation defined inductively on its syntax
  - computes a *fold* over a program's AST
- ***Big-step abstract machine***
  - Execution traces instead of trees
- ***Small-step operational semantics***
  - Each step: decompose-contract-recompose
- ***Reduction semantics***
  - Contexts and contractions
- ***Small-step abstract machine***
  - Examples: CC, SCC, CK, CEK-machines, Krivine's machine, Landin's SECD etc.

Related work: Ager-al:PPDP03, Cardelli:TR107, Cousineau-al:SCP87, Danvy:IFL04, Danvy:ICFP08, Felleisen-Friedman:FDPC3, Hannan-Miller:MSCS92, Krivine:04, Landin:CJ64, Launchbury:POPL93, Milne-Strachey:76, Plotkin:JLAP04, Reynolds:ACM72, Sestoft:JFP97, VanHorn-Might:ICFP10...

A diversity of semantic artifacts

Semantics are described via some *meta-languages*

A diversity of semantic artifacts

Semantics are described via some *meta-languages*

Any expressive *programming language*
may play a role of a *meta-language*

A diversity of semantic artifacts

Semantics are described via some *meta-languages*

Any expressive *programming language*
may play a role of a *meta-language*

therefore

Semantic formalisms can be directly represented as ***programs***
(or *semantic artifacts*)

Semantics equivalence problem

All these semantics were developed independently of each other

Semantics equivalence problem

All these semantics were developed independently of each other

Their equivalence should be proved

Semantics equivalence problem

All these semantics were developed independently of each other

Their equivalence should be proved

Can we connect them some other way?

Calculational inter-derivation

Program *semantics* can be connected
via the inter-derivation of the corresponding *semantic artifacts*.[1]

[1] O. Danvy, ICFP '08

This connection has never been done
for *type checking*

Outline

Type checking: description I

Type checking is a semantics of a "typing language" on top of the host language's syntax.

Type checking: description I

Type checking is a semantics of a "typing language" on top of the host language's syntax.

Its natural semantics is a familiar one, in the form of *derivation rules*.

$$\text{[t-lam]} \quad \frac{\Gamma[x:\tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2} \qquad \text{[t-var]} \quad \frac{(x : \tau \in \Gamma)}{\Gamma \vdash x : \tau}$$

$$\text{[t-app]} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1 e_2 : \tau_2} \qquad \text{[t-num]} \quad \Gamma \vdash \mathit{number} : \text{num}$$

Type system for the simply typed lambda calculus with numbers

Type checking: description II

Another semantics of typing language: a *small-step abstract machine with control and result stacks* (SEC-machine)[2]

$$
\begin{array}{rcl}
\langle S,\ E,\ num{:}C\rangle & \Rightarrow_t & \langle \mathsf{num}{:}S,\ E,\ C\rangle \\
\langle S,\ E[x{\Rightarrow}\tau],\ x{:}C\rangle & \Rightarrow_t & \langle \tau{:}S,\ E[x{\Rightarrow}\tau],\ C\rangle \\
\langle S,\ E,\ (\lambda x{:}\tau.e){:}C\rangle & \Rightarrow_t & \langle \mathsf{nil},\ E\sqcup\{x{\Rightarrow}\tau\},\ e{:}Lam(\tau,\ S){:}C\rangle \\
\langle S,\ E,\ (e_1 e_2){:}C\rangle & \Rightarrow_t & \langle S,\ E,\ e1{:}Fun(e_2){:}C\rangle \\
\langle \tau_2{:}S,\ E,\ Lam(\tau_1,\ S'){:}C\rangle & \Rightarrow_t & \langle (\tau_1{\rightarrow}\tau_2){:}S',\ E,\ C\rangle \\
\langle (\tau_1{\rightarrow}\tau_2){:}S,\ E,\ Fun(e_2){:}C\rangle & \Rightarrow_t & \langle (\tau_1{\rightarrow}\tau_2){:}S,\ E,\ e_2{:}Arg(\tau_1,\ \tau_2){:}C\rangle \\
\langle \tau_1{:}x{:}S,\ E,\ Arg(\tau_1,\ \tau_2){:}C\rangle & \Rightarrow_t & \langle \tau_2{:}S,\ E,\ C\rangle
\end{array}
$$

[2] C. Hankin and D. Le Métayer, POPL ’94

Type checking: description II

Another semantics of typing language: a *small-step abstract machine with control and result stacks* (SEC-machine)[2]

$$
\begin{array}{rcl}
\langle S,\ E,\ num{:}C\rangle & \Rightarrow_t & \langle \mathsf{num}{:}S,\ E,\ C\rangle \\
\langle S,\ E[x{\Rightarrow}\tau],\ x{:}C\rangle & \Rightarrow_t & \langle \tau{:}S,\ E[x{\Rightarrow}\tau],\ C\rangle \\
\langle S,\ E,\ \boxed{(\lambda x{:}\tau.e)}\ {:}C\rangle & \Rightarrow_t & \langle \mathsf{nil},\ \boxed{E\sqcup\{x{\Rightarrow}\tau\}}\ ,\ e{:}\ \boxed{Lam(\tau,\ S)}\ {:}C\rangle \\
\langle S,\ E,\ (e_1 e_2){:}C\rangle & \Rightarrow_t & \langle S,\ E,\ e1{:}Fun(e_2){:}C\rangle \\
\langle\ \boxed{\tau_2}\ {:}S,\ E,\ \boxed{Lam(\tau_1,\ S')}\ {:}C\rangle & \Rightarrow_t & \langle\ \boxed{(\tau_1{\rightarrow}\tau_2)}\ {:}S',\ E,\ C\rangle \\
\langle(\tau_1{\rightarrow}\tau_2){:}S,\ E,\ Fun(e_2){:}C\rangle & \Rightarrow_t & \langle(\tau_1{\rightarrow}\tau_2){:}S,\ E,\ e_2{:}Arg(\tau_1,\ \tau_2){:}C\rangle \\
\langle \tau_1{:}x{:}S,\ E,\ Arg(\tau_1,\ \tau_2){:}C\rangle & \Rightarrow_t & \langle \tau_2{:}S,\ E,\ C\rangle
\end{array}
$$

[2] C. Hankin and D. Le Métayer, POPL '94

Type checking: description III

And yet another one: *reduction semantics*[3]

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid \lambda x{:}\tau.e \mid e\ e \mid \tau \rightarrow e \mid \textsf{num} \\
T & ::= & T\ e \mid \tau\ T \mid \tau \rightarrow T \mid [\ ] \\
\tau & ::= & \textsf{num} \mid \tau \rightarrow \tau \\
n & ::= & \textit{number}
\end{array}
$$

Hybrid language and type-checking contexts

$$
\begin{array}{rcll}
T[n] & \mapsto_t & T[\textsf{num}] & [\textsf{tc-const}] \\
T[\lambda x{:}\tau.e] & \mapsto_t & T[\tau \rightarrow \{\tau/x\}\ e] & [\textsf{tc-lam}] \\
T[(\tau_1 \rightarrow \tau_2)\ \tau_1] & \mapsto_t & T[\tau_2] & [\textsf{tc-}\tau\beta]
\end{array}
$$

Type-checking reduction rules

[3] G. Kuan, D. MacQueen and R. B. Findler, ESOP '07

Why should we care about different semantics?

Given formalism for a type systems defines a corresponding *semantic artifact*

Why should we care about different semantics?

Given formalism for a type systems defines a corresponding *semantic artifact*

- Type derivation rules $\sim$ recursive descent
- Machine-like semantics $\sim$ driver-loop abstract machine (CEK, SECD etc.)
- Reduction semantics $\sim$ *decompose-contract-recompose* loop

Why should we care about different semantics?

Given formalism for a type systems defines a corresponding *semantic artifact*

- Type derivation rules $\sim$ recursive descent
- Machine-like semantics $\sim$ driver-loop abstract machine (CEK, SECD etc.)
- Reduction semantics $\sim$ *decompose-contract-recompose* loop

Benefits of non-standard semantics:

- type debugging
- optimized computation

Outline

Semantics equivalence again

Do all these semantics describe
the same type checking procedure?

Semantics equivalence again

Do all these semantics describe
the same type checking procedure?

Theorem *[Hankin and Le Métayer]*
(Soundness and Completeness for $\Rightarrow_t$)
$\Gamma \vdash e : \tau$ iff $\langle S, \Gamma, e : C \rangle \Rightarrow_t \langle \tau : S, \Gamma, C \rangle$.

Theorem *[Kuan et al.] (Soundness and Completeness for $\mapsto_t$)*
For any e and τ, $\emptyset \vdash e : \tau$ iff $e \mapsto_t^* \tau$

Our concern

Can we *inter-derive* these semantics **a priori**
rather than *prove* their equivalence **a posteriori?**

Our contribution

Yes, we can.

Our contribution

Yes, we can.

via *functional inter-derivation*

Transformations instead of proofs

Outline

Two approaches

A mathematician's approach: to prove the equivalence between semantics by induction (or bisimulation by coinduction)

A programmer's approach:

- take one particular implementation
- apply a series of transformations to a program
- be sure that transformations are correct

Two approaches

A mathematician's approach: to prove the equivalence between semantics by induction (or bisimulation by coinduction)

A programmer's approach:

- take one particular implementation
- apply a series of transformations to a program
- be sure that transformations are correct

All transformations are already proved to be correct

A toolbox

Semantic-preserving functional program transformations

A toolbox

Semantic-preserving functional program transformations

- continuation-passing style transform
  (Plotkin, Steele, Friedman, Wand, Danvy, Filinski)

A toolbox

Semantic-preserving functional program transformations

- continuation-passing style transform
  (Plotkin, Steele, Friedman, Wand, Danvy, Filinski)
- defunctionalization (Reynolds)

A toolbox

Semantic-preserving functional program transformations

- continuation-passing style transform
  (Plotkin, Steele, Friedman, Wand, Danvy, Filinski)
- defunctionalization (Reynolds)
- explicit control stack introduction (Landin, Danvy)

A toolbox

Semantic-preserving functional program transformations

- continuation-passing style transform
  (Plotkin, Steele, Friedman, Wand, Danvy, Filinski)
- defunctionalization (Reynolds)
- explicit control stack introduction (Landin, Danvy)

All transformations are proved to be correct

A toolbox

Semantic-preserving functional program transformations

- continuation-passing style transform
  (Plotkin, Steele, Friedman, Wand, Danvy, Filinski)
- defunctionalization (Reynolds)
- explicit control stack introduction (Landin, Danvy)

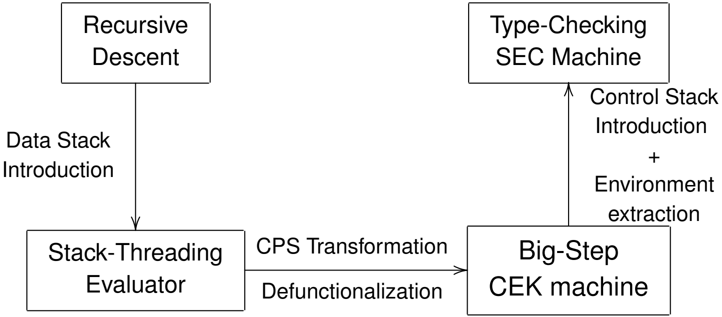
All transformations are proved to be correct

Each one yields a new adequate representation of the algorithm

Inter-derivation

This work: inter-derivation between a *recursive descent* and an *abstract machine*.

Our goal is SEC-machine[4].



[4] *Result* ***S****tack* $\times$ ***E****nvironment* $\times$ ***C****ontrol Stack*

Outline

Summary and conclusion

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;
- Different formalisms and corresponding implementations might be used, but ***equivalence*** between them *should be proved*;

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;
- Different formalisms and corresponding implementations might be used, but ***equivalence*** between them *should be proved*;
- Functional correspondence by program transformations enables us to ***derive*** a family of algorithms for type checking, rather than ***invent*** them from scratch;

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;
- Different formalisms and corresponding implementations might be used, but ***equivalence*** between them *should be proved*;
- Functional correspondence by program transformations enables us to ***derive*** a family of algorithms for type checking, rather than ***invent*** them from scratch;
- A tool-chain of transformations is applied to derive those algorithms;

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;
- Different formalisms and corresponding implementations might be used, but ***equivalence*** between them *should be proved*;
- Functional correspondence by program transformations enables us to ***derive*** a family of algorithms for type checking, rather than ***invent*** them from scratch;
- A tool-chain of transformations is applied to derive those algorithms;
- All derived algorithms are correct

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;
- Different formalisms and corresponding implementations might be used, but ***equivalence*** between them *should be proved*;
- Functional correspondence by program transformations enables us to ***derive*** a family of algorithms for type checking, rather than ***invent*** them from scratch;
- A tool-chain of transformations is applied to derive those algorithms;
- All derived algorithms are correct ***by construction***.

Summary and conclusion

- Type checking is a computation over a program's syntax; its semantics may be described in different ways;
- Different formalisms and corresponding implementations might be used, but ***equivalence*** between them *should be proved*;
- Functional correspondence by program transformations enables us to ***derive*** a family of algorithms for type checking, rather than ***invent*** them from scratch;
- A tool-chain of transformations is applied to derive those algorithms;
- All derived algorithms are correct ***by construction***.

Thank you