Type-preserving compilation via dependently typed syntax in Agda

Andreas Abel[1]

[1]Department of Computer Science and Engineering
Chalmers and Gothenburg University, Sweden

Types for Proofs and Programs (TYPES 2020)
Torino, Italy
Scheduled for 2-5 March
(Cancelled due to COVID-19)

Verified Compilation

- Tony Hoare's Grand Challenge: Verified compilation.
- CompCert for the masses?
- Full verification may be too expensive ($> 90\%$ of impl. effort).
- Sweet spot: lots of confidence for little verification.
- Compiler be a total function.

Verifying Type-Safety

- Robin Milner: *Well-typed programs do not go wrong.*
- Types checked by compiler front-end.
- Goal: preserve properties through back-end.
  - Type safety.
  - "Execution safety": No illegal jumps.
- Typed machine language (e.g. LLVM).

Method

- Implement compiler in a dependently-typed programming language.
- Represent well-typed syntax as indexed data types.
- Type-correct compilation enforced by indexing discipline.

Intrinsically well-typed syntax

object language	**meta language**
untyped e. g.: syntax trees	simply typed e. g.: (C, Java), Scala, ML, Haskell, ...
simply typed e. g.: λ-calculus, C--	dependently typed e. g.: Agda, Coq, Idris, Lean, ...
dependently typed	dependently typed

Pipeline

$$
\begin{array}{c}
\texttt{C--}\ \text{text} \\
\big\downarrow\ \text{parser} \\
\text{abstract syntax} \\
\big\downarrow\ \text{type checker} \\
\text{well-typed syntax} \\
\big\downarrow\ \text{code generator} \\
\text{well-formed machine code} \\
\big\downarrow\ \text{printer} \\
\texttt{JVM--}\ \text{symbolic assembler}
\end{array}
$$

C-- by example

```
// Does p divide q?
bool divides (int p, int q) {
  return (q / p) * p == q;
}

// Is p prime?
bool prime (int p) {
  if (p <= 2) return p == 2;
  else {
    int q = 3;
    while (q * q <= p)
      if (divides(q,p)) return false;
      else q = q + 2;
  }
  return true;
}
```

C-- language elements

- Hierarchical:
  - function definitions contain statements,
  - statements contain expressions.
- Types: $\mathsf{Ty} = \{\mathsf{int}, \mathsf{double}, \mathsf{bool}, \mathsf{void}\}$.
- Variables (function parameters, local variables) are *scoped*.
- Some statements declare new variables (`int q = 3;`).
- Control structures: `if`, `while`, `return`.

Typing contexts

- Scoping is managed by *typing contexts* Γ, snoc-lists of types.
- Example list $\mathsf{int}^2 = [\mathsf{int}, \mathsf{int}]$:

$$\varepsilon.\mathsf{int}.\mathsf{int}$$

- Category Cxt:
  - Objects: typing contexts Γ.
  - Morphisms $\Gamma \subseteq \Delta$ are ways in which Γ is a sublist of Δ.

$$\mathsf{skip}\ \frac{as \subseteq bs}{as \subseteq (bs.b)} \qquad \mathsf{keep}\ \frac{as \subseteq bs}{(as.a) \subseteq (bs.a)} \qquad \mathsf{done}\ \frac{}{\varepsilon \subseteq \varepsilon}$$

- Variables (de Bruijn indexes) pick a type from a context.

$$\mathsf{Var}_t\,\Gamma \cong ([t] \subseteq \Gamma)$$

- Quiz:
  1. How many morphisms in $\mathsf{int}^2 \subseteq \mathsf{int}^5$?
  2. How many morphisms in $\mathsf{int}^k \subseteq \mathsf{int}^n$?

Cxt has only weak push-outs

$$
\begin{tikzcd}
{[]} \arrow[r] \arrow[d] & {[a]} \arrow[d] \arrow[ddr] & \\
{[b]} \arrow[r] \arrow[drr] & {[a, b]} & \\
 & & {[b, a]}
\end{tikzcd}
$$

Well-typed syntax

- $\mathsf{Var}_t\,\Gamma$ variables of type t
- $\mathsf{Exp}_t\,\Gamma$ expressions of type t
- $\mathsf{Stm}_r\,\Gamma\,\Gamma'$ statements
  - r: return type of function
  - Γ: context before statement
  - $\Gamma' = \Gamma.\Delta$: context after
- $\mathsf{Stms}_r\,\Gamma\,\Gamma'$ statement sequences: free category over Stm.

Expressions

- $\mathsf{Exp}_t : \mathsf{Cxt} \to \mathsf{Set}$ functor
- maps hom $\eta : \Gamma \subseteq \Delta$ to weakening $[\eta] : \mathsf{Exp}_t\,\Gamma \to \mathsf{Exp}_t\,\Delta$
- constructors

$$
\begin{array}{lcll}
\mathsf{lit} & : & (v : \mathsf{Val}_t) & \to \mathsf{Exp}_t\,\Gamma \\
\mathsf{var} & : & (x : \mathsf{Var}_t\,\Gamma) & \to \mathsf{Exp}_t\,\Gamma \\
\mathsf{arith} & : & (op : \mathsf{ArithOp}\,t)\ (e_1\ e_2 : \mathsf{Exp}_t\,\Gamma) & \to \mathsf{Exp}_t\,\Gamma \\
\mathsf{cmp} & : & (op : \mathsf{CmpOp}\,t)\ (e_1\ e_2 : \mathsf{Exp}_t\,\Gamma) & \to \mathsf{Exp}_{\mathsf{bool}}\,\Gamma
\end{array}
$$

Statements

$$
\begin{array}{lll}
\mathsf{assign} & : (x : \mathsf{Var}_t\,\Gamma)\ (e : \mathsf{Exp}_t\,\Gamma) & \to \mathsf{Stm}_r\,\Gamma\,\Gamma \\
\mathsf{decl} & : (t : \mathsf{Ty}) & \to \mathsf{Stm}_r\,\Gamma\,(\Gamma.t) \\
\mathsf{return} & : (e : \mathsf{Exp}_r\,\Gamma) & \to \mathsf{Stm}_r\,\Gamma\,\Gamma \\
\mathsf{while} & : (e : \mathsf{Exp}_{\mathsf{bool}}\,\Gamma)\ (s : \mathsf{Stm}_r\,\Gamma\,\Gamma') & \to \mathsf{Stm}_r\,\Gamma\,\Gamma \\
\mathsf{if} & : (e : \mathsf{Exp}_{\mathsf{bool}}\,\Gamma)\ (s_1 : \mathsf{Stm}_r\,\Gamma\,\Gamma_1)\ (s_2 : \mathsf{Stm}_r\,\Gamma\,\Gamma_2) & \to \mathsf{Stm}_r\,\Gamma\,\Gamma
\end{array}
$$

Java Virtual Machine (JVM)

- no registers
- stack for evaluating expressions
- local variable store (incl. function parameters)
- (heap for objects)
- method call handling behind the scenes

Java Virtual Machine (JVM) example

C--	Jasmin (symbolic JVM)

```
bool divides
   (int p, int q)
{
  return
  (q / p) * p == q;
}
```

```
.method divides(II)I
        iload_1                ;; q
        iload_0                ;; p
        idiv
        iload_0                ;; p
        imul

        iload_1                ;; q
        if_icmpeq L_true
        iconst_0               ;; false
        goto L_done
L_true: iconst_1               ;; true
L_done: ireturn
.end method
```

Evaluation Stack

- JVM has local stack for evaluation of expressions.
- Stack type $\mathsf{ST} = \mathsf{List}\,\mathsf{Ty}$
- Stack instruction $\mathsf{SI}_\Gamma\,\Phi\,\Phi'$
  - $\Gamma : \mathsf{Cxt}$ local variable store typing
  - $\Phi : \mathsf{ST}$ stack typing before instruction
  - $\Phi' : \mathsf{ST}$ stack typing after
- Constructors:

$$
\begin{array}{lclll}
\mathsf{ldc} & : & (i : \mathsf{Val}_{\mathsf{int}}) & \to \mathsf{SI}_\Gamma\,\Phi & (\Phi.\mathsf{int}) \\
\mathsf{load} & : & (x : \mathsf{Var}_t\,\Gamma) & \to \mathsf{SI}_\Gamma\,\Phi & (\Phi.t) \\
\mathsf{store} & : & (x : \mathsf{Var}_t\,\Gamma) & \to \mathsf{SI}_\Gamma\,(\Phi.t) & \Phi \\
\mathsf{arith} & : & (op : \mathsf{ArithOp}\,t) & \to \mathsf{SI}_\Gamma\,(\Phi.t.t) & (\Phi.t)
\end{array}
$$

- Instruction sequences $\mathsf{SIS}_\Gamma\,\Phi\,\Phi'$: free category over SI_Γ.

Variable typing administration

- Variable declarations $\mathsf{decl}\, t : \mathsf{Stm}\, \Gamma\, (\Gamma.t)$ are NOPs.
- Needed in intrinsically typed machine language.

$$\mathsf{decl}\, t : (\Gamma, \Phi) \to (\Gamma.t, \Phi)$$

- Reconstruction in actual JVM by static analysis (bytecode verifier).
- Machine type $\mathsf{MT} = \mathsf{Cxt} \times \mathsf{ST}$.

Jumps can go wrong

- Bad jump:

```
                                 ;; Stack modification:
         if_icmpeq L_true        ;; [int,int] -> []
         iconst_0                ;; []        -> [int]
L_true:  istore_3                ;; [int]     -> []
```

- Jump target needs to have same stack typing as source.
- Same for variable typing.
- Labels are typed by "before" machine type Ξ of target.
- Label context $\mathsf{Labels} = \mathsf{List}\,\mathsf{MT}$.
- A label is a de Bruijn index $\ell : \mathsf{Label}_{\Xi}\,\Lambda$.

$$\mathsf{Label}_{\Xi}\,\Lambda \cong ([\Xi] \subseteq \Lambda)$$

Jump targets need to exist

- Semantics of a label is the code following it.
- Each label needs to point to some code.
- Two types of labels:
  - *Join points* for branches of if are lets.
  - *Back jumps* to repeat body of while are fixs.

Join points: let

```
[[ if (e) s1; else s2; s ]] =

  let l  = [[s]]
      l1 = [[s1]]; goto l
      l2 = [[s2]]; goto l
  in  [[e]]; branch l1 l2
```

Back jumps: fix

```
[[ while (e) s0; s ]] =

  let l2 = [[s]]
  in  fix l.
        let l1 = [[s0]]; goto l
        in  [[e]]; branch l1 l2
```

Flowchart (control flow graph)

- $\mathsf{FC}_r\,\Xi\,\Lambda$ control flow graph
  - r return type of method
  - Ξ machine state on entry
  - Λ typed jump targets
- Constructors:

$$
\begin{array}{llllll}
\mathsf{exec} & : (i : \mathsf{SI}_\Gamma\,\Phi\,\Phi') & (fc : \mathsf{FC}_r\,(\Gamma,\Phi')\,\Lambda) & \to \mathsf{FC}_r\,(\Gamma,\Phi) & \Lambda \\
\mathsf{decl} & : (t : \mathsf{Ty}) & (fc : \mathsf{FC}_r\,(\Gamma.t,\varepsilon)\,\Lambda) & \to \mathsf{FC}_r\,(\Gamma,\varepsilon) & \Lambda \\
\mathsf{return} & : (e : \mathsf{Exp}_r\,\Gamma) & & \to \mathsf{FC}_r\,(\Gamma,\varepsilon) & \Lambda \\
\mathsf{goto} & : (\ell : \mathsf{Label}_\Xi\,\Gamma) & & \to \mathsf{FC}_r\,\Xi & \Lambda \\
\mathsf{branch} & : (o : \mathsf{CmpOp}\,t) & (fc\ fc' : \mathsf{FC}_r\,(\Gamma,\Phi)\,\Lambda) & \to \mathsf{FC}_r\,(\Gamma,\Phi.t.t) & \Lambda \\
\mathsf{let} & : (fc' : \mathsf{FC}_r\,\Xi'\,\Lambda) & (fc : \mathsf{FC}_r\,\Xi\,(\Lambda.\Xi')) & \to \mathsf{FC}_r\,\Xi & \Lambda \\
\mathsf{fix} & : & (fc : \mathsf{FC}_r\,\Xi\,(\Lambda.\Xi)) & \to \mathsf{FC}_r\,\Xi & \Lambda
\end{array}
$$

Back end

- Code generation: translation from well-typed syntax to flow chart using continuations.
- Linearization: from flowcharts to basic blocks.
- Printing: from basic blocks to Jasmin symbolic JVM.

Evaluation

- *When it type-checks, it works.*
- Had only 3 bugs in compiler on first run!
- Agda programming requires hard thinking ahead.
- Little proof effort.
- Too hard for average beginning master student.
- Full verification in progress:
  - Needs reasoning in sublist-category.
  - Contributed categorical constructions (e.g. weak pushout) to Agda standard library.

Related Work

- Andrew Appell, Modern compiler implementation in C/Java/ML
- Xavier Leroy et al., CompCert, in Coq
- Magnus Myreen et al., CakeML, in HOL
- DeepSpec project: Verified tool chain
- Greg Morrisett et al., Typed Assembly Language
- Alberto Pardo, Emmanuel Gunter, Miguel Pagano, Marcos Viera, *An Internalist Approach to Correct-by-Construction Compilers*, PPDP'18: Terms indexed by semantics (in Agda)