Type-preserving compilation
via dependently typed syntax in Agda

Andreas Abel

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**

<table>
<thead>
<tr>
<th>object language</th>
<th>meta language</th>
</tr>
</thead>
<tbody>
<tr>
<td>untyped</td>
<td>simply typed</td>
</tr>
<tr>
<td>e. g.: syntax trees</td>
<td>e. g.: (C, Java), Scala, ML, Haskell, ...</td>
</tr>
<tr>
<td>simply typed</td>
<td>dependently typed</td>
</tr>
<tr>
<td>e. g.: (\lambda)-calculus, C--</td>
<td>e. g.: Agda, Coq, Idris, Lean, ...</td>
</tr>
<tr>
<td>dependently typed</td>
<td>dependently typed</td>
</tr>
</tbody>
</table>
Pipeline

C-- text
  ↓ parser
  ↓ abstract syntax
  ↓ type checker
  ↓ well-typed syntax
  ↓ code generator
  ↓ well-formed machine code
  ↓ printer
  ↓ JVM-- symbolic assembler
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: \( Ty = \{ \text{int}, \text{double}, \text{bool}, \text{void} \} \).
- Variables (function parameters, local variables) are scoped.
- Some statements declare new variables (\texttt{int q = 3;}).
- Control structures: if, while, return.
Typing contexts

- Scoping is managed by *typing contexts* \( \Gamma \), snoc-lists of types.
- Example list \( \text{int}^2 = [\text{int, int}] \):

  \[
  \varepsilon.\text{int}\text{.int}
  \]

- Category \( \text{Cxt} \):
  - Objects: typing contexts \( \Gamma \).
  - Morphisms \( \Gamma \subseteq \Delta \) are ways in which \( \Gamma \) is a sublist of \( \Delta \).

  \[
  \text{skip} \quad \frac{as \subseteq bs}{as \subseteq (bs.b)} \quad \text{keep} \quad \frac{as \subseteq bs}{(as.a) \subseteq (bs.a)} \quad \text{done} \quad \frac{}{\varepsilon \subseteq \varepsilon}
  \]

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

  \[
  \text{Var}_t \Gamma \cong ([t] \subseteq \Gamma)
  \]

- Quiz:
  1. How many morphisms in \( \text{int}^2 \subseteq \text{int}^5 \)?
  2. How many morphisms in \( \text{int}^k \subseteq \text{int}^n \)?
Cxt has only weak push-outs

\[
\begin{array}{c}
\emptyset \\
\downarrow \\
[b]
\end{array} \quad \rightarrow 
\begin{array}{c}
[a] \\
\downarrow \\
[a, b]
\end{array} \\
\begin{array}{c}
[b] \\
\downarrow \\
[a, b]
\end{array} \quad \rightarrow 
\begin{array}{c}
[b, a]
\end{array}
\]
Well-typed syntax

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

- \( \text{Exp}_t : \text{Cxt} \to \text{Set} \) functor
- maps hom \( \eta : \Gamma \subseteq \Delta \) to weakening \( [\eta] : \text{Exp}_t \Gamma \to \text{Exp}_t \Delta \)
- constructors

\[
\begin{align*}
\text{lit} & : (v : \text{Val}_t) \quad \to \text{Exp}_t \Gamma \\
\text{var} & : (x : \text{Var}_t \Gamma) \quad \to \text{Exp}_t \Gamma \\
\text{arith} & : (op : \text{ArithOp}_t) (e_1 \ e_2 : \text{Exp}_t \Gamma) \quad \to \text{Exp}_t \Gamma \\
\text{cmp} & : (op : \text{CmpOp}_t) (e_1 \ e_2 : \text{Exp}_t \Gamma) \quad \to \text{Exp}_{\text{bool}} \Gamma
\end{align*}
\]
Statements

\[
\begin{align*}
\text{assign} & : (x : \text{Var}_t \Gamma) \ (e : \text{Exp}_t \Gamma) \quad \rightarrow \quad \text{Stm}_r \Gamma \Gamma \\
\text{decl} & : (t : \text{Ty}) \quad \rightarrow \quad \text{Stm}_r \Gamma (\Gamma. t) \\
\text{return} & : (e : \text{Exp}_r \Gamma) \quad \rightarrow \quad \text{Stm}_r \Gamma \Gamma \\
\text{while} & : (e : \text{Exp}_\text{bool} \Gamma) \ (s : \text{Stm}_r \Gamma \Gamma') \quad \rightarrow \quad \text{Stm}_r \Gamma \Gamma \\
\text{if} & : (e : \text{Exp}_\text{bool} \Gamma) \ (s_1 : \text{Stm}_r \Gamma \Gamma_1) \ (s_2 : \text{Stm}_r \Gamma \Gamma_2) \quad \rightarrow \quad \text{Stm}_r \Gamma \Gamma
\end{align*}
\]
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--

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

Jasmin (symbolic JVM)

.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 $ST = \text{List Ty}$
- Stack instruction $SI_\Gamma \Phi \Phi'$
  - $\Gamma : \text{Cxt}$ local variable store typing
  - $\Phi : ST$ stack typing before instruction
  - $\Phi' : ST$ stack typing after
- Constructors:

\[
\begin{align*}
\text{ldc} & : (i : \text{Val}_{\text{int}}) \rightarrow SI_\Gamma \Phi \quad (\Phi.\text{int}) \\
\text{load} & : (x : \text{Var}_t \Gamma) \rightarrow SI_\Gamma \Phi \quad (\Phi.t) \\
\text{store} & : (x : \text{Var}_t \Gamma) \rightarrow SI_\Gamma (\Phi.t) \Phi \\
\text{arith} & : (op : \text{ArithOp}_t) \rightarrow SI_\Gamma (\Phi.t.t)(\Phi.t)
\end{align*}
\]

- Instruction sequences $SIS_\Gamma \Phi \Phi'$: free category over $SI_\Gamma$. 

Abel

Dependently-Typed Compilation

TYPES 2020 16 / 25
Variable typing administration

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

$$\text{decl } t : (\Gamma, \Phi) \rightarrow (\Gamma.t, \Phi)$$

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

- Bad jump:

```java
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 $\Xi$ of target.
- Label context $\text{Labels} = \text{List MT}$.
- A label is a de Bruijn index $\ell : \text{Label}_\Xi \Lambda$.

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

- $\text{FC}_r \Xi \Lambda$ control flow graph
  - $r$ return type of method
  - $\Xi$ machine state on entry
  - $\Lambda$ typed jump targets

- Constructors:

  - `exec`: $(i : \text{SI}_\Gamma \Phi \Phi') (fc : \text{FC}_r (\Gamma, \Phi') \Lambda) \rightarrow \text{FC}_r (\Gamma, \Phi) \Lambda$
  - `decl`: $(t : \text{Ty}) (fc : \text{FC}_r (\Gamma.t, \varepsilon) \Lambda) \rightarrow \text{FC}_r (\Gamma, \varepsilon) \Lambda$
  - `return`: $(e : \text{Exp}_r \Gamma) \rightarrow \text{FC}_r (\Gamma, \varepsilon) \Lambda$
  - `goto`: $(\ell : \text{Label}_\Xi \Gamma) \rightarrow \text{FC}_r \Xi \Lambda$
  - `branch`: $(o : \text{CmpOp} t) (fc fc' : \text{FC}_r (\Gamma, \Phi) \Lambda) \rightarrow \text{FC}_r (\Gamma, \Phi.t.t) \Lambda$
  - `let`: $(fc' : \text{FC}_r \Xi' \Lambda) (fc : \text{FC}_r \Xi (\Lambda.\Xi')) \rightarrow \text{FC}_r \Xi \Lambda$
  - `fix`: $(fc : \text{FC}_r \Xi (\Lambda.\Xi)) \rightarrow \text{FC}_r \Xi \Lambda$
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