From Formal Correctness to Engineering Validity

Giuseppe Primiero

UNIVERSITÀ DEGLI STUDI DI MILANO
DIPARTIMENTO DI FILOSOFIA
“PIERO MARTINETTI”

PROGRAMme Workshop
Bertinoro
1. The Formal Background

2. Formal Milestones

3. Principles of Formal Correctness

4. Physical Computing

5. Principles of Physical Validity
1. The Formal Background

2. Formal Milestons

3. Principles of Formal Correctness

4. Physical Computing

5. Principles of Physical Validity
Curry-Howard-de Bruijn Correspondence

First observed by [Curry (1934)], then later independently reformulated by [Howard (1969)] and [de Bruijn (1970)] in the 60s and 70s respectively:

Principle (Curry-Howard-de Bruijn Correspondence)

Propositions are Types.
Early Steps: Gentzen I

\[
\begin{align*}
\frac{}{A \vdash A} & \quad \text{Id} \\
\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \land B} & \quad \land\text{-intro} \\
\frac{\Gamma \vdash A \land B}{\Gamma \vdash A} & \quad \land\text{-elim(left)} \\
\frac{\Gamma, A \vdash B}{\Gamma \vdash A \lor B} & \quad \lor\text{-intro(left)} \\
\frac{\Gamma \vdash A \lor B \quad \Delta \vdash A \lor C}{\Gamma, \Delta \vdash A \lor C} & \quad \lor\text{-elim} \\
\frac{\Gamma \vdash A \lor B \quad \Delta \vdash B \lor C}{\Gamma, \Delta \vdash C} & \quad \lor\text{-elim}
\end{align*}
\]
[Curry (1934)]: a function $f : A \rightarrow B$ with an input of type $A$ and an output $B$ can be intuitively read as a corresponding implication $A \supset B$ and hence for any computable function there is a provable proposition of that form, and vice versa.

[Howard (1969)], making use of results by [Prawitz (1965)]: now every natural deduction pair of rules can be reformulated in the typed $\lambda$-calculus.
Early Steps: Typed Church I

\[ x : A \vdash x : A \quad \text{Id} \]
\[ \Gamma \vdash t : A \quad \Delta \vdash u : B \]
\[ \Gamma, \Delta \vdash \langle t, u \rangle : A \land B \quad \land \text{–intro} \]
\[ \Gamma \vdash t : A \land B \]
\[ \Gamma \vdash \pi_1(t) : A \quad \land \text{–elim(left)} \]
\[ \Gamma \vdash t : A \land B \]
\[ \Gamma \vdash \pi_2(t) : B \quad \land \text{–elim(right)} \]
\[ \Gamma, x : A \vdash t : B \]
\[ \Gamma \vdash \lambda x \cdot t : A \supset B \quad \supset \text{–intro} \]
\[ \Gamma \vdash t : A \]
\[ \Delta \vdash u : A \supset B \]
\[ \Gamma, \Delta \vdash t(u) : B \quad \supset \text{–elim} \]
\[ \Gamma \vdash t : A \land B \]
\[ \Gamma \vdash \lambda x \cdot t(u) \mid \lambda y \cdot t(v) : C \quad \lor \text{–intro(left)} \]
\[ \Gamma \vdash t : A \lor B \]
\[ \Gamma, x : A \vdash u : C \]
\[ \Gamma, y : B \vdash v : C \]
\[ \Gamma \vdash \lambda x \cdot t(u) \mid \lambda y \cdot t(v) : C \quad \lor \text{–elim} \]

Primiero
16 October 2018 7/47
<table>
<thead>
<tr>
<th>FUNCTION TYPE</th>
<th>PROPOSITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \rightarrow B$</td>
<td>$A \supset B$</td>
</tr>
<tr>
<td>$A \times B$</td>
<td>$A \land B$</td>
</tr>
<tr>
<td>$A + B$</td>
<td>$A \lor B$</td>
</tr>
<tr>
<td>$\lambda(b) : \Pi(A, B)$</td>
<td>$(\forall x : A)B(x)$</td>
</tr>
<tr>
<td>$\langle a, b \rangle : \Sigma(A, B)$</td>
<td>$(\exists x : A)B(x)$</td>
</tr>
</tbody>
</table>

**Figure:** Correspondence between function types and propositions
Correspondence revisited

Principle (Curry-Howard-de Bruijn Correspondence – Revisited)

Proofs are Programs.
<table>
<thead>
<tr>
<th>Programming</th>
<th>Mathematics</th>
</tr>
</thead>
<tbody>
<tr>
<td>program, procedure, algorithm</td>
<td>function</td>
</tr>
<tr>
<td>input</td>
<td>argument</td>
</tr>
<tr>
<td>output, result</td>
<td>value</td>
</tr>
<tr>
<td>$x := e$ (assignment)</td>
<td>$x = e$</td>
</tr>
<tr>
<td>$S_1; S_2$</td>
<td>composition of functions</td>
</tr>
<tr>
<td>if $B$ then $S_1$ else $S_2$</td>
<td>definition by cases</td>
</tr>
<tr>
<td>while $B$ do $S$</td>
<td>definition by recursion</td>
</tr>
<tr>
<td>data structure</td>
<td>element by recursion</td>
</tr>
<tr>
<td>data type</td>
<td>set, type</td>
</tr>
<tr>
<td>value of data type</td>
<td>element of a set, object of a type</td>
</tr>
<tr>
<td>$a : A$</td>
<td>$a \in A$</td>
</tr>
<tr>
<td>integer</td>
<td>$Z$</td>
</tr>
<tr>
<td>real</td>
<td>$R$</td>
</tr>
<tr>
<td>boolean</td>
<td>${0, 1}$</td>
</tr>
<tr>
<td>$(c_1, \ldots, c_n)$</td>
<td>${c_1, \ldots, c_n}$</td>
</tr>
<tr>
<td>array $[I]$ of $T$</td>
<td>$T^I, I \rightarrow T$</td>
</tr>
<tr>
<td>record $s_1 : T_1; s_2 : T_2$ end</td>
<td>$T_1 \times T_2$</td>
</tr>
<tr>
<td>record case $s : (c_1, c_2)$ of $c_1 : (s_1 : T_1); c_2 : (s_2 : T_2)$ end</td>
<td>$T_1 + T_2$</td>
</tr>
<tr>
<td>set of $T$</td>
<td>${0, 1}^T, T \rightarrow {0, 1}$</td>
</tr>
</tbody>
</table>

**Figure:** Key notions of Programming with mathematical counterparts
Correctness

Definition (Program Correctness Problem)

Given any program $p$, is it possible to prove whether $p$ executed on the appropriate input $i$ returns the intended output $o$, or in other words whether it satisfies its intended function?
1. The Formal Background

2. Formal Milestones

3. Principles of Formal Correctness

4. Physical Computing

5. Principles of Physical Validity
Floyd, [Floyd (1967)]

- An association of steps in the control flow of a program with logically holding propositions.

- A semantic definition of the program syntactically specifies linguistic constructs expressing valid commands and their conditions.

- An interpretation corresponds to a mapping of logical sentences to each transition from step to step (called the entry and exit of the command).

- A verification of the flowchart of the program is a proof that for every command expressed in it, if there is an entrance such that a given sentence is true, then there must be an exit such that another sentence is true.
Figure 1. Flowchart of program to compute $S = \sum_{j=1}^{n} a_j \ (n \geq 0)$
Termination and Machine-independence

“we have not proved that an exit will ever be reached; the methods so far described offer no security against nonterminating loops. ”

Properties of programs in a given language are independent of their physical instantiation and processors for that language, and can be reduced to “establishing standards of rigor for proofs about programs in the language”.
Hoare’s Axiomatic Semantics, [Hoare (1969)]

- An *assertion* corresponds to a predicate describing the state of a program at any point of its execution.

- Determine the valid assignments of these variables *before* program execution, its *preconditions*.

- Check that the program execution leads to the expected set of variable assignments or program states *after* the program operations have been performed, its *postconditions*.

\[ P\{Q\}R \]
Figure: Rules of Floyd-Hoare Logic
Definition (Hoare Correctness)

A program is correct with respect to a pre- and post-condition specification if the execution of the program on a machine whose initial state satisfies the pre-condition will result in the machine terminating in a state which satisfies the post-condition.
No side-effects and Universality

“*The axioms and rules of inference quoted in this paper have implicitly assumed the absence of side effects of the evaluation of expressions and conditions.*”

The axiomatic method is *general*, implementations can be designed so that they should be required to satisfy the axioms, and it is *universally applicable*, so that aspects of the program – like data-types – can be left underspecified.
Definition (Weakest Pre-Condition)

The weakest pre-condition, denoted $wp(P, R)$ is the condition that characterizes the set of all initial states such that execution of program $P$ will result with certainty in a final state satisfying a given post-condition $R$.

The semantics of $P$ is given by determining for any post-condition $R$ the corresponding weakest pre-condition $wp(P, R)$. If the initial state does not satisfy a weakest pre-condition, $P$ may not establish the truth of $R$ in its final state, or $P$ may fail to reach a final state at all, because it enters an infinite loop or it gets stuck,[Dijkstra (1975), p.17].
Dijkstra: Total Correctness

**Definition (Total Program Correctness)**

For a given specification \( (\text{pre}(P), R) \), a program \( P \) is totally correct with respect to its specification iff \( \text{pre}(P) \rightarrow \text{wp}(P, R) \)
Definition (Partial Program Correctness)

A program $P$ is partially correct if it can be proved that
\[ pre(P) \land wp(P, true) \rightarrow wp(P, R), \]
where $wp(P, true)$ is the weakest pre-condition ensuring termination in any state.
Correctness vs. Efficiency

“most techniques for proving the correctness of a program treat the program text as a static, rather formal, mathematical object that can be dealt with independently of the fact that there may exist machines that could execute such a program. As such, a clear separation of concerns emerges: we might call them the mathematical concerns about correctness and the engineering concerns about efficiency. In contrast to the correctness concerns, the efficiency concerns are only meaningful in relation to implementations and it is only during the efficiency concerns that we need remember that the program text is intended to evoke computational processes. Both the mathematical and the engineering concerns have, of course, always been with us, but once they used to be dealt with inextricably intertwined.”
The Formal Background

Formal Milestones

Principles of Formal Correctness

Physical Computing

Principles of Physical Validity
Principle (Algorithmic dependence)

*Computation is a mathematical process in which an algorithmic description is lingustically encoded.*
Principle (Encoding independence)

The output of a given equivalence class of computational processes is unaffected by the properties of the chosen linguistic encoding when these processes are considered at the highest level of abstraction.
Definition (Valid Formal Computation)

A computational process $p$ is formally valid if and only if any expression $e$ required by the linguistic encoding of the algorithmic description of $p$ is logically admissible and the process described by $p$ terminates in a state whose linguistic encoding expresses its intended output $S$. 
1 The Formal Background

2 Formal Milestones

3 Principles of Formal Correctness

4 Physical Computing

5 Principles of Physical Validity
Resource Aware Logic I

\[ \phi \vdash \phi \] \text{Identity}

\[ \{\} \vdash 1 \] \text{I-}\text{I}

\[ \Gamma(\{\}m) \vdash \chi \quad \Delta \vdash 1 \] \text{I-}\text{E}

with \( \{\} \) the multiplicative unit and 1 the empty resource (in the multiplicative fragment).

\[ \Gamma, \phi \vdash \psi \] \
\[ \Gamma \vdash \phi \rightarrow \psi \] \text{→-I}

\[ \Gamma \vdash \phi \rightarrow \star \psi \quad \Delta \vdash \phi \] \text{→-E}

\[ \Gamma \vdash \phi \quad \Delta \vdash \psi \] \text{→-I}

\[ \Gamma, \Delta \vdash \phi \star \psi \] \text{→-E}

\[ \Gamma(\{\}a) \vdash \chi \quad \Delta \vdash 1 \] \text{I-}\text{E}

with \( \{\} \) the additive unit and 1 the empty resource (in the additive fragment).

**Figure:** Rules of BI Logic, [O’Hearn, Pym (1999)]
Formal Correctness

Definition (Partial Correctness)

\{ \phi \} C \{ \psi \} is true if for all \( R = \langle s, h \rangle \) it is the case that \( R \models \phi \) implies that \( \langle C, R \rangle \) is safe and if there is a transformation according to \( C \) to a memory configuration \( R' = \langle s', h' \rangle \), then \( R' \models \psi \).
**Definition (Partial Correctness)**

$\{\phi\} C \{\psi\}$ is true if for all $R = \langle s, h \rangle$ it is the case that $R \models \phi$ implies that $\langle C, R \rangle$ is safe and if there is a transformation according to $C$ to a memory configuration $R' = \langle s', h' \rangle$, then $R' \models \psi$.

**Definition (Total Correctness)**

$\{\phi\} C \{\psi\}$ is true if for all $R = \langle s, h \rangle$ it is the case that $R \models \phi$ implies that $\langle C, R \rangle$ terminates normally and if there is a transformation according to $C$ to a memory configuration $R' = \langle s', h' \rangle$, then $R' \models \psi$. 
The resources from which a program depends need to be actually accessible/available at run time;

the program terminates with the intended specification under all possible configuration of resources.
**Definition (Consistency)**

Every read process receives the most recent write process, or an error.

**Definition (Availability)**

Every request process receives a (non-error) response – without guarantee that it contains the most recent write process.

**Definition (Partition Tolerance)**

The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.
Theorem (CAP Theorem)

One can only practically build a distributed system exhibiting any two characteristics among Consistency, Availability and Partition-tolerance.
Partiality

1. either processes are consistent and live, but the system may fail in the presence of dropped messages

2. or is consistent and persistent, but may contain error responses

3. or is persistent in the presence of dropped messages and live even in the presence of older messages, but then it might not preserve consistency.
1 The Formal Background

2 Formal Milestones

3 Principles of Formal Correctness

4 Physical Computing

5 Principles of Physical Validity
Principles of Physical Computational Validity

1. **Functionality**: the ability of a physically constructed computational system to satisfy the aims and objectives for which it has been designed, possibly notwithstanding the presence of non-essential errors.

2. **Efficiency**: the principle that functionalities need to be delivered at least as fast as required, parametrized by computational resources, and possibly preferred to fully correct computations which are not efficient.

3. **Usability**: the principle that practical interaction with the system guarantees both service delivery and resource usage within admissible limits.
Principle (Architectural Dependence)

A physical computation is the implementation in a physical infrastructure of the linguistic encoding of an algorithmic process.
A physical computation may behave differently depending on the architecture’s properties.
Definition (Functional Validity)

A physical computing artefact \( t \) is functionally valid if and only if the functionalities of its intended specification \( S \) are logically admissible.
**Definition (Functional Correctness)**

A physical computing artefact $t$ is functionally correct if and only if the functionalities of its intended specification $S$ are displayed with the intended efficiency even in the presence of negligible errors when $t$ is used in the appropriate way.
A physical artefact $t$ performs a valid computation if and only if

1. $t$ is functionally valid (i.e. its intended specification $S$ is logically admissible according to an algorithmic description $p$), and

2. the information processing $c$ implementing the linguistic encoding $e$ of $p$ is functionally correct and efficient (and possibly optimally so), and

3. $c$ will terminate in a state satisfying $S$. 
Final observations

- Formal correctness requires principles of algorithmic dependence and encoding independence.

- Physical validity of computing requires principles of architectural and behavioural dependence.

- Their co-existence is allowed by an analysis that includes the different levels of abstraction of a computational system.
Thanks.
References


References II


References IV

