Illi Isabellistes Se Custodes Egregios Praestabant

Simon Bischof, Joachim Breitner, Denis Lohner, and Gregor Snelting

Abstract We present two new results in machine-checked formalizations of programming languages. 1. *Probabilistic Noninterference* is a central notion in software security analysis. We present the first Isabelle formalization of low-security observational determinism ("LSOD"), together with a proof that LSOD implies probabilistic noninterference. The formalization of LSOD uses a *flow-sensitive* definition of low-equivalent traces, which drastically improves precision. 2. We present the first full and machine-checked proof that Launchbury's well-known semantics of the lazy lambda-calculus is correct as well as adequate. The proof catches a bug in Launchbury's original proof, which was open for many years.
Both results continue the work of the "Quis Custodiet" project at KIT, which aims at machine-checked soundness proofs for complex properties of languages, compilers, and program analysis. We thus include a short overview of earlier "Quis Custodiet" results.

1 Introduction

"*Quis custodiet ipsos custodes?*"[1] is the motto of a long-standing project at KIT and TUM, where Isabelle is used to verify complex properties of programming lan-

Simon Bischof
Karlsruhe Institute of Technology (KIT), e-mail: simon.bischof@kit.edu

Joachim Breitner
University of Pennsylvania, e-mail: joachim@cis.upenn.edu

Denis Lohner
Karlsruhe Institute of Technology (KIT), e-mail: denis.lohner@kit.edu

Gregor Snelting
Karlsruhe Institute of Technology (KIT), e-mail: gregor.snelting@kit.edu

[1] "Who will guard the guards?", a question by the roman satirist Juvenal, ca. 100 A.C.

guages, compilers, and program analysis methods. At KIT, “Quis Custodiet” provided the following major results:

- The semantics of multiple inheritance in C++ was formalized in Isabelle, and type safety was proven [36, 33];
- Program dependence graphs were formalized in Isabelle, and PDG-based information flow control – a specific form of software security analysis – was proven to be sound [35, 34];
- The Java memory model was completely formalized in Isabelle, and sequential consistency was proven; this project included a full semantics for threads and a verified Java compiler [19, 18, 17];
- A new code optimization for Haskell, called call arity, was implemented, proven correct and integrated into the GHC [3, 5, 4].

All together, the “Quis Custodiet” project at KIT produced over 100000 lines of Isabelle code, distributed over 14 AFP publications.

In this contribution, we present two new results in the “Quis Custodiet” project:

1. *Probabilistic Noninterference* (PN) is a central notion in software security. In particular, information flow control (IFC) algorithms check program code for confidentiality and integrity leaks, and guarantee noninterference. We present a formalization of the well-known low-security observational determinism IFC criterion (“LSOD”), which is flow-sensitive and thus much more precise than previous approaches. We provide an Isabelle proof that our flow-sensitive LSOD implies PN.
2. We present the first full and machine-checked proof that Launchbury’s well-known semantics of the lazy λ-calculus is correct as well as adequate. In particular, Launchbury’s original adequacy proof had a bug, which could not be fixed for many years. Using Isabelle, a new proof approach was discovered, which allowed a machine-checked adequacy proof.

Together with our earlier work, we can thus answer the original question“*Quis custodiet ipsos custodes?*” by stating: “*Illi Isabellistes se custodes egregios praestabant!*”.[2]

2 Providing Software Security Guarantees: Isabelle Soundness Proofs for Noninterference

2.1 Background

“Quis Custodiet” was originally founded with the goal to provide machine-checked soundness proofs for certain software security analysis algorithms, in particular algorithms for *information flow control* (IFC). IFC analyses program code, and checks

[2] We leave the translation as an exercise to the reader.

it for *confidentiality* (“no secret values can leak to public ports”) and *integrity* (“critical computations cannot be manipulated from outside”). A *sound* IFC guarantees that all potential leaks are discovered, while a *precise* IFC does not cause false alarms.[3] To prove soundness of an IFC analysis, the notion of *noninterference* is essential [25]. In particular, for multi-threaded programs subtle leaks resulting from scheduling and nondeterminism must be found resp. prohibited, which requires that IFC guarantees *probabilistic noninterference* (PN) [26].

Many authors investigated properties and variations of PN definitions, and some built IFC tools that check program code for confidentiality leaks based on PN. At KIT, Snelting et al. developed the JOANA system, which can handle full Java with unlimited threads, scales to 250 kLOC, guarantees PN, produces few false alarms, has a nice GUI, and is open source [12]. JOANA achieves scalability and precision by using sophisticated program analysis technology, such as program dependence graphs (PDGs), points-to analysis, exception analysis, and may-happen-in-parallel analysis. In particular, the analysis is flow-sensitive, context-sensitive, and object-sensitive, which drastically improves precision [13]. JOANA was successfully used to guarantee confidentiality of an experimental e-voting system [15], and to analyse the full source of the HSQLDB database [11].

JOANA can handle unlimited threads and provides a new algorithm for PN. This “RLSOD” algorithm is more precise than competing methods, while avoiding limitations or soundness bugs of previous algorithms [7, 10, 1]. RLSOD is based on the classical “low-security observational determinism” (LSOD) approach, but, for the first time, allows secure low-nondeterminism. RLSOD again exploits modern program analysis, thus being much more precise than LSOD [1].

In the scope of “Quis Custodiet”, PDGs and the PDG-based sequential noninterference were formalised in Isabelle, and a machine-checked soundness proof was provided [35]. For PN however, noninterference and its analysis are more demanding. While soundness proofs for PN checkers based on *security type systems* have successfully been provided [22, 26], the PDG-based approach is, due to its flow- and context-sensitivity, much more complex to formalize. A machine-checked soundness proof for RLSOD has just begun. As a first step, the next sections describe the Isabelle formalization of flow-sensitive LSOD and its PN guarantee. To keep this article self-contained, we begin with a summary of technical PN properties.

2.2 Technical Basics

IFC guarantees that no violations of confidentiality or integrity may occur. For confidentiality, all values in input, output, or program states are classified as “high”

[3] Note that “100% soundness + 100% precision” cannot be achieved simultaneously: the famous Rice Theorem states that such perfect program analysis is undecidable.

```
1  void main():
2    read(H);
3    if (H < 1234)
4       print(0);
5    L = H;
6    print(L);
```

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    read(L);
6    print(L);
7  void thread_2():
8    read(H);
9    L = H;
```

```
1  void main():
2    fork thread_1();
3    fork thread_2();
4  void thread_1():
5    longCmd();
6    print("AR");
7  void thread_2():
8    read(H);
9    while (H != 0)
10      H--;
11   print("ND");
```

Fig. 1 Some leaks. Left: explicit and implicit, middle: possibilistic, right: probabilistic. For simplicity, we assume that `read(L)` reads low variable `L` from a low input channel; `print(H)` prints high variable `H` to a high output channel.

(secret) or "low" (public), and it is assumed that an attacker can read all low values, but cannot see any high value.[4]

Figure 1 presents small but typical confidentiality leaks. As usual, variable `H` is "High" (secret), `L` is "Low" (public). Explicit leaks arise if (parts of) high values are copied (indirectly) to low output. Implicit leaks arise if a high value can change control flow, which can change low behaviour (see Fig. 1 left). Possibilistic leaks in concurrent programs arise if a certain interleaving produces an explicit or implicit leak; in Fig. 1 middle, interleaving order 5, 8, 9, 6 causes an explicit leak. Probabilistic leaks arise if the probability of low output is influenced by high values. For example in Fig. 1 right, there are no explicit or implicit leaks; but if the value of `H` is large, probability is higher that "`ARND`" is printed instead of "`NDAR`". Thus the attacker may gather information about `H` from public output.

The simplest (sequential) noninterference definition assumes that a global and static classification $cl(v)$ of all program variables v as secret (H) or public (L) is given. The attacker can only see public variables and values. Noninterference then requires that for any two initial state with identical L variables, but perhaps varying H variables, the final states also coincide on L variables. Thus an attacker cannot learn anything about H variables, and confidentiality is guaranteed.[5]

Technically, the simplest form of (sequential) noninterference is defined as follows. Let $\mathscr{P}$ be a program. Let s, s' be initial program states, let $[\![\mathscr{P}]\!](s)$, $[\![\mathscr{P}]\!](s')$ be the final states after executing $\mathscr{P}$ in state s and s', resp. Noninterference holds iff

$$s \sim_L s' \implies [\![\mathscr{P}]\!](s) \sim_L [\![\mathscr{P}]\!](s') \ .$$

[4] A more detailed discussion of IFC attacker models can be found in, e.g., [10]. Note that JOANA allows arbitrary lattices of security classifications, not just the simple $\bot = L \leq H = \top$ lattice. Note also that integrity is dual to confidentiality, but will not be discussed here; JOANA can handle both.

[5] Note that noninterference covers security only on the program level, it does not cover side channel attacks, compromised hardware, etc. The latter must be handled by other security techniques.

The relation $s \sim_L s'$ means that two states are low-equivalent, that is, coincide on low variables: $cl(v) = L \implies s(v) = s'(v)$. Program input is assumed to be part of the initial states s, s', and program output is assumed to be part of the final states ("batch-mode behaviour").

In multi-threaded programs, fine-grained interleaving effects must be accounted for, thus traces are used instead of states. A trace is a (possibly infinite) sequence of events $t = (s_0, o_0, s_1), (s_1, o_1, s_2), \ldots, (s_\nu, o_\nu, s_{\nu+1}), \ldots$, where the o_ν are operations (i.e. dynamically executed program statements), and s_ν resp. $s_{\nu+1}$ are the program states before resp. after execution of o_ν.

For PN, the notion of low-equivalent traces is essential. In the simplest definition, traces t, t' are low-equivalent if for all low-observable events $(s_\nu, o_\nu, s_{\nu+1}) \in t$, $(s'_\nu, o'_\nu, s'_{\nu+1}) \in t'$ it holds that $s_\nu \sim_L s'_\nu$, $o_\nu = o'_\nu$, and $s_{\nu+1} \sim_L s'_{\nu+1}$. For low-equivalent traces, we write $t \sim_L t'$. Obviously, $\sim_L$ is an equivalence relation, and the low-class of t is $[t]_L = \{t' \mid t' \sim_L t\}$. Note that the $t' \in [t]_L$ cannot be distinguished by an attacker, as all $t' \in [t]_L$ have the same public behaviour. Thus $[t]_L$ represents t's low behaviour.

PN is called "probabilistic" because it essentially depends on the probabilities for certain traces under certain inputs. Probabilistic behaviour is caused by program nondeterminism (e.g., races), scheduler behaviour, interleaving, and other factors. We write $P_i(T)$ for the probability that a trace $t \in T$ is executed under input i. Thus, $P_i([t]_L)$ is the probability that some trace $t' \sim_L t$ is executed under i. In practice, the $P_i([t]_L)$ are very difficult or impossible to determine – fortunately, for our soundness proof explicit probabilities are not required.

The following PN definition uses explicit input streams instead of initial states. For both inputs the same initial state is assumed. Inputs consist of a low and a high stream of values. Inputs are low-equivalent ($i \sim_L i'$) if their low streams are equal.

Now let i and i' be inputs; let $T(i)$ be the set of all possible traces of program $\mathscr{P}$ for input i. Obviously, we have $P_i(T(i)) = 1$. In the following definition, we use $P_i([t]_L \cap T(i))$ instead of $P_i([t]_L)$. Note that $[t]_L \setminus T(i)$ is a subset of all impossible traces, which is a null set for P_i. Let $\Theta = T(i) \cup T(i')$. PN holds iff

$$\begin{aligned} i \sim_L i' \implies \forall t \in \Theta :\ & [t]_L \cap T(i) \text{ is measurable for } P_i \\ & \wedge\ [t]_L \cap T(i') \text{ is measurable for } P_{i'} \\ & \wedge\ P_i([t]_L \cap T(i)) = P_{i'}([t]_L \cap T(i')) \ . \end{aligned}$$

That is, if we take any trace t which can be produced by i or i', the probability that a $t' \in [t]_L$ is executed is the same under i resp. i'. In other words, **probability for any public behaviour is independent from the choice of i or i'** and thus cannot be influenced by secret input. Note that for the equality of probabilities to have a meaning, both sets must be measurable. It is easy to prove that PN implies sequential noninterference, as the proof is independent of specific P_i (see [7, 1]).

Applying this to Fig. 1 right, we first observe that all inputs are low-equivalent as there is only high input. For any $t \in \Theta$ there are only two possibilities:
`...print("AR")...print("ND")...`$\in t$ or
`...print("ND")...print("AR")...`$\in t$.

Thus, there are only two equivalence classes
$[t]_L^1 = \{t' \mid \ldots\texttt{print("AR")}\ldots\texttt{print("ND")}\ldots \in t'\}$ and
$[t]_L^2 = \{t' \mid \ldots\texttt{print("ND")}\ldots\texttt{print("AR")}\ldots \in t'\}$.
Now if i contains a small value and i' a large value, as discussed earlier we have $P_i([t]_L^1) \neq P_{i'}([t]_L^1)$ as well as $P_i([t]_L^2) \neq P_{i'}([t]_L^2)$, hence PN is violated.

LSOD is the oldest and simplest criterion which enforces PN. LSOD is independent from specific P_i, and thus independent of specific scheduler behaviour. This attractive feature made us choose LSOD as a starting point for JOANA's PN. LSOD demands that low-equivalent inputs produce low-equivalent traces. It is intuitively secure: changes in high input can never change low behaviour, because low behaviour is enforced to be deterministic. This is however a very restrictive requirement. To address this, RLSOD, as used in JOANA, relaxed this restriction and led to a powerful and precise analysis [7, 1].

Technically, let i, i' be inputs, Θ as above. LSOD holds iff

$$i \sim_L i' \implies \forall t, t' \in \Theta : t \sim_L t' \quad .$$

Under LSOD, all traces t for input i are low-equivalent: $T(i) \subseteq [t]_L$, because $\forall t' \in T(i) : t' \sim_L t$. If there is more than one trace for i, then this must result from high-nondeterminism; low behaviour is strictly deterministic.

Lemma 1. *LSOD implies PN.*

Proof. Let $i \sim_L i', t \in \Theta$. W.l.o.g. let $t \in T(i)$.

Due to LSOD, we have $T(i) \subseteq [t]_L$ and thus $[t]_L \cap T(i) = T(i)$. As $P_i(T(i)) = 1$, the set $[t]_L \cap T(i)$ is measurable. Therefore, we have

$$P_i([t]_L \cap T(i)) = P_i(T(i)) = 1.$$

Likewise, the set $[t]_L \cap T(i')$ is measurable and $P_{i'}([t]_L \cap T(i')) = 1$, so we have $P_i([t]_L \cap T(i)) = P_{i'}([t]_L \cap T(i'))$. □

The proof makes obvious that LSOD does not care for specific P_i – which is both its strength (scheduler independence) and its weakness (lack of precision).

2.3 Giffhorn's Flow Sensitive LSOD

The above simple definition for $\sim_L$ is extremely restrictive and causes many false alarms. Not only does it require an unrealistic "lock-step" execution of both traces, it is also based on a static H/L classification of program variables, which is neither flow- nor context- nor object-sensitive.

Using security type systems, more refined definitions of $\sim_L$ have been investigated (see, e.g., [26, 37, 24]). In 2012, Giffhorn discovered that PDGs can be used to define low-equivalent traces in such a way that a) lock-step execution is not necessary; b) infinite traces are covered, and soundness problems of earlier approaches to infinite traces are avoided; c) PN checking is flow-, context- and object-sensitive. This insight fits nicely with JOANA's PDG-based IFC. In the following, we sketch Giffhorn's approach; more details and explanations can be found in [10].

Giffhorn observed that due to flow sensitivity, the same variable or memory cell can have different classifications at different program points resp. corresponding trace operations (because it has multiple occurrences in the PDG) without compromising soundness.[6] For a low event (s,o,s'), its (flow-sensitive!) low projection is $(s\mid_{use(o)},o,s'\mid_{def(o)})$, where $s\mid_{use/def(o)}$ denotes the restriction of s to the variables read and written in operation o, resp.[7] Giffhorn allows infinite traces and defines low-equivalent traces as follows:

Let t,t' be two traces. Let t_L,t'_L be their low-observable behaviours, which are obtained by deleting high events and using the low projections for low events. Let $t_L[n]$ be the n-th event in the low-observable behaviour of t, $k_t = |t_L|, k_{t'} = |t'_L|$. Then $t \sim_L t'$ iff

1. $\forall 0 \leq i < \min(k_t,k_{t'}) : t_L[i] = t'_L[i]$, and
2. if $k_t \neq k_{t'}$, and w.l.o.g. $k_t > k_{t'}$, then t' is infinite and $\forall k_{t'} \leq j < k_t$: t' infinitely delays an operation $o \in DCD(op(t_L[j]))$, where $op(e)$ is the operation of event e.

The latter condition is new and makes sure that operations $op(t_L[j])$ missing in the shorter low-observable trace t'_L can only be missing due to nontermination before o, where o is transitively dynamically control dependent (DCD) on $op(t_L[j])$. [10] explains a) that this subtle definition avoids soundness problems known from earlier definitions of low-equivalent infinite traces; b) that dynamic control dependence can be soundly and precisely approximated through PDGs. Giffhorn thus provides a static, PDG-based criterion which guarantees LSOD and thus PN. We will not go into the details of the static criterion and its soundness proof (see [10, 9]). Instead we now present an Isabelle proof that LSOD based on Giffhorn's $\sim_L$ guarantees PN.

2.4 Giffhorn's LSOD in Isabelle

The Isabelle formalization of Giffhorn's approach starts with definitions and lemmas about traces, postdominance, and dynamic control dependency (*DCD*). In fact, all events in a trace can be uniquely determined by their chain of dynamic control predecessors, which is expressed by a series of lemmas. Note there are no concrete definitions for dynamic control dependency and postdomination, but just minimal requirements, which can be instantiated in various ways. Next, low observable events and low-equivalence of traces are defined. Giffhorn's crucial innovation, the condition "t' infinitely delays $o \in DCD(o_j)$" reads as follows:

definition *infinite-delay* :: *'trace* ⇒ *'stmt operation* ⇒ *bool* (- *id* -)

[6] Note that in JOANA only input and output must be classified, all other classifications are computed automatically by a fixpoint operation [13].

[7] This flow-sensitive definition increases precision, see [10].

where $T\ id\ o' \equiv \neg\ o' \in_t T \wedge (\exists o''.\ \textit{in-same-branch}\ o'\ o'' \wedge o'' \in_t T)$
where *in-same-branch* uses *DCD* chains to check if o', o'' are in the same branch, caused by some branching point b (e.g., a dynamic IF statement). If so, but $o' \notin_t T$, then U infinitely delays o' due to nontermination between b and o' [10].

Low-equivalency $\approx_{low}$ follows Giffhorn's original definition (see [10], definition 6). The final *LSOD* definition reads

locale $LSOD =$
...
fixes $\textit{input-low-eq} :: \textit{'input} \Rightarrow \textit{'input} \Rightarrow \textit{bool}$ (**infix** $\sim_L$ *50*)
and $\textit{possible-traces} :: \textit{'input} \Rightarrow \textit{'trace set}$ $(T'(\text{-}'))$

definition $\Theta\ i\ i' \equiv T(i) \cup T(i')$
definition *LSOD*
$\equiv \forall i\ i'.\ i \sim_L i' \longrightarrow (\forall t \in \Theta\ i\ i'.\ \forall t' \in \Theta\ i\ i'.\ t \approx_{low} t')$

Note that *input-low-eq* is a parameter of locale *LSOD* and can again be instantiated in various ways. To prove that Giffhorn's LSOD implies PN, we originally planned to use the formalization of PN in Isabelle as described by Popescu and Nipkow [23]. Unfortunately, this work assumes that $\approx_{low}$ is an equivalence relation, while Giffhorn's $\approx_{low}$ is not transitive (more precisely, it is transitive only in the finite case). Therefore, we defined our own notion of PN in Isabelle using the *HOL-Probability* library [14].

We define PN in an abstract setting (independent of *LSOD*), that gets instantiated later. In this setting we assume the existence of *probability spaces* P_i and sets of possible traces $T(i)$ for each input i such that $P_i(T(i)) = 1$. As before, the definition is parametric in *input-low-eq*, and this time also in the low relation between traces.

locale *Probabilistic-Noninterference* $=$
fixes $\textit{input-prob} :: \textit{'input} \Rightarrow \textit{'trace measure}$ $(P_{\text{-}})$
and $\textit{input-low-eq} :: \textit{'input} \Rightarrow \textit{'input} \Rightarrow \textit{bool}$ (**infix** $\sim_L$ *50*)
and $\textit{low-rel} :: \textit{'trace} \Rightarrow \textit{'trace} \Rightarrow \textit{bool}$ (**infix** $\approx_{low}$ *50*)
and $\textit{possible-traces} :: \textit{'input} \Rightarrow \textit{'trace set}$ $(T'(\text{-}'))$
assumes *prob-space*: *prob-space* P_i
and *prob-T*: $P_i\ T(i) = 1$

The PN definition then reads

definition $PN \equiv \forall i\ i'.\ i \sim_L i'$
$\longrightarrow (\forall t \in \Theta\ i\ i'.$
$\{t' \in T(i).\ t \approx_{low} t'\} \in \textit{sets}\ P_i \wedge$
$\{t' \in T(i').\ t \approx_{low} t'\} \in \textit{sets}\ P_{i'} \wedge$
$P_i\ \{t' \in T(i).\ t \approx_{low} t'\} = P_{i'}\ \{t' \in T(i').\ t \approx_{low} t'\})$

where *sets* P_i denotes the set of measurable sets of P_i.

Connecting the abstract PN definition to *LSOD* is straightforward by equating the possible traces of both formalizations and instantiating *low-rel* with Giffhorn's low-equivalency. The proof of the soundness theorem

theorem *LSOD-implies-PN*: **assumes** *LSOD* **shows** *PN*

follows the handwritten proof in Lemma 1 (we would like to point out that Isabelle's Sledgehammer tool was quite helpful in finding the right lemmas from the probability library). The reader should keep in mind that this soundness theorem is only the first step for an RLSOD soundness proof in Isabelle. For a manual proof of RLSOD soundness, see [1].

To justify the assumptions made by the abstract settings above, we adopted Popescu and Nipkow's [23] construction of the trace space using a discrete time markov chain (see [14]) and instantiated *input-prob* of our locale *Probabilistic-Noninterference* with the resulting trace space[8].

Further, we formalized *multi-threaded CFGs* as a set of CFGs (as presented by Wasserrab [33]) extended with fork edges[9], for which we defined a multithreaded small-steps semantics. The state is modelled as a tuple consisting of the memory and for each thread a list of the executed edges – that implies, that there is no thread local memory. We then define the semantics to be the intra-thread semantics (that is annotated to the edges, see [33]) on each thread, spawning new threads for each fork edge.

As there are only finitely many threads in each state, we can compute a discrete probability distribution for the successor states of a given state and use that as the transition distribution (see [14]) for our discrete time markov chain. For that we used *pmf-of-multiset* on the multi-set of possible successor states, thus modelling a uniform scheduler.

In addition to the trace semantics of the discrete time markov chain we defined a second trace semantics using possibly infinite lists (type *'a llist*) and traditional interleaving of threads to model finite and infinite executions explicitly. We then proved the two semantics to be equivalent[10].

We instantiated the multi-threaded CFGs with two simple multi-threaded toy languages, a structured while language, and a goto language, both extended with a fork operation.

The complete formalization of flow-sensitive LSOD and PN, including parallel CFGs, dominance, dynamic dependences, and the toy languages comprises about 10 kLOC of Isabelle text[11].

3 An Old Proof Completed, Finally

Formal semantics provide the underpinning for proofs in language-based security, as without them, many theorems cannot even be stated, let alone be proved in a rig-

[8] In contrast to Popescu and Nipkow, our traces include the initial state.

[9] We currently don't support any form of synchronization primitives such as join/wait.

[10] Note that in the case of finite traces, the discrete time markov chain remains in the finished state forever.

[11] The complete formalization can be found at `http://pp.ipd.kit.edu/git/LSOD/`.

$$\frac{}{\Gamma : \lambda x.\, e \Downarrow \Gamma : \lambda x.\, e}\textsc{Lam} \qquad \frac{\Gamma : e \Downarrow \Delta : \lambda y.\, e' \qquad \Delta : e'[y{:=}x] \Downarrow \Theta : v}{\Gamma : e\, x \Downarrow \Theta : v}\textsc{App}$$

$$\frac{\Gamma : e \Downarrow \Delta : v}{x \mapsto e, \Gamma : x \Downarrow x \mapsto v, \Delta : v}\textsc{Var} \qquad \frac{\mathsf{dom}\,\Delta \cap \mathsf{fv}(\Gamma) = \{\} \qquad \Delta, \Gamma : e \Downarrow \Theta : v}{\Gamma : \textbf{let}\ \Delta\ \textbf{in}\ e \Downarrow \Theta : v}\textsc{Let}$$

Fig. 2 Launchbury's natural semantics for Lazy Evaluation

orous, machine-checked manner. All (standard) semantics describe mathematically what a program does, but they vary in style (e.g., operational big-step semantics vs. denotational semantics) and in detail (i.e. which details of the actual program execution are modelled and which are abstracted over).

Launchbury's seminal paper "A Natural Semantics for Lazy Evaluation" [16] introduces two such semantics for a lambda calculus with lazy evaluation:

1. An operational semantics where the relation $\Gamma : e \Downarrow \Delta : v$ indicates that the expression e, evaluated within the heap Γ, evaluates to the value v while changing the heap to Δ, where a heap is modeled as a partial map from variable names to expressions. The relation is inductively defined by the rules in Fig. 2.
   Because the shape of the heap is explicit in this formulation, the semantics allows us to represent expressions as graphs instead of trees, and thus express sharing, lazy evaluation and memory usage. This is indeed relevant for an IFC analysis of functional programming languages with lazy evaluation, as sharing can cause hidden channels [8].
2. A denotational semantics $[\![e]\!]$ which translates an expression e into a mathematical object (an element of a cpo, to be precise) that captures its functional behaviour. This semantics is more abstract than the operational, as it does not model the evolution of the heap.

In order to prove that the operational semantics makes sense, and to allow transferring results from one semantics to the other, they are connected by two theorems: *Correctness* states that given $\Gamma : e \Downarrow \Delta : v$, the denotational semantics assigns the same meaning to $\textbf{let}\ \Gamma\ \textbf{in}\ e$ and $\textbf{let}\ \Delta\ \textbf{in}\ v$. *Adequacy* states that every program that has a meaning under the denotational semantics (i.e. $[\![e]\!] \neq \bot$) also evaluates in the operational semantics.

3.1 A Sketchy History

Launchbury's paper comes with a detailed proof of correctness, which translates nicely into an Isabelle proof. For the proof of adequacy, the story is not quite so simple.

$$\frac{\Gamma : e \Downarrow \Delta : \lambda y.\, e' \qquad y \mapsto x, \Delta : e' \Downarrow \Theta : v}{\Gamma : e\, x \Downarrow \Theta : v}\textsc{App}' \qquad \frac{x \mapsto e, \Gamma : e \Downarrow \Delta : v}{x \mapsto e, \Gamma : x \Downarrow \Delta : v}\textsc{Var}'$$

Fig. 3 Launchbury's alternative natural semantics

In his paper, Launchbury outlines an adequacy proof via two intermediate semantics:

1. He introduces the variant ANS of his operational semantics which differs in three aspects (Fig. 3):

   a. In the rule for function application $(\lambda y.\, e)\ x$, instead of substituting x for y in e, the binding $y \mapsto x$, called an *indirection*, is added to the heap.
   b. In the rule for evaluating a variable x, the original semantics removes the binding for $x \mapsto e$ from the heap until e is evaluated. This models the *black-holing* technique that is employed by language implementations such as the Haskell compiler GHC in order to detect some forms of divergence. It also allows the garbage collector to free unused memory in a more timely manner [20]. The alternative operational semantics does not perform black-holing.
   c. Also in the rule for evaluating a variable x that is bound to e, after evaluating e to v, the original semantics binds x to v, so that further uses of x will not recalculate this value. This *updating* is essential to lazy evaluation, and differentiates it from the call-by-name evaluation strategy, where e would be re-evaluated for every use of x. Again, the alternative operational semantics omits this step.

   The intention is that these changes make ANS behave more similarly to the denotational semantics and thus simplify the adequacy proof.
2. He introduces a variant of the denotational semantics $\mathcal{N}[\![e]\!]$, dubbed the *resourced denotational semantics*, which expects as an additional parameter an element of $C := \mathbb{N} \cup \omega$, and decreases this argument in every recursive call.
   When the argument is ω, this does not actually limit the number of recursive calls, and the semantics coincide to the regular denotational semantics.
   If the argument is a natural number, the semantics might run out of steam before it fully calculates the meaning of the given expression.

Launchbury's adequacy proof sketch then proceeds with little more detail than the following list:

1. If $[\![e]\!] \neq \bot$, then, because the semantics coincide, we have $\mathcal{N}[\![e]\!]\ \omega \neq \bot$.
2. Because the resourced denotational semantics are continuous, there is an $n \in \mathbb{N}$ such that $\mathcal{N}[\![e]\!]\ n \neq \bot$.
3. From that we can show that in the alternative operational semantics, there exist Δ and v such that $\{\} : e \Downarrow \Delta : v$. The step is performed by induction on n, and justifies the introduction of the resourced denotational semantics.

4. Finally, because the two operational semantics obviously only differ in the shape of the heap, but otherwise reduce the same expressions, we also have such a derivation in the original operational semantics.

Given the impact of the paper as the default big-step operational semantics for lazy evaluation, it is not surprising that there are later attempts to work out the details of this proof sketch. These turned up a few hurdles.

The first hurdle is that it does not make sense to state the coincidence of the denotational semantics as $[\![e]\!] = \mathscr{N}[\![e]\!]\,\omega$, since these semantics map into different cpos. The regular denotational semantics maps into the cpo D defined by the domain equation

$$D = [D \to D]_\perp$$

where every element is either $\perp$ or a continuous function from D to D. The resourced denotational semantics map into the cpo E defined by the domain equation

$$E = [(C \to E) \to (C \to E)]_\perp \enspace .$$

This problem was first discovered by Sánchez-Gil *et al.*, and just fixing this step, by defining a suitable similarity relation between these cpos, was a notable contribution on its own [27].

The same group also attempted to complete the other steps of Launchbury's proof sketch, in particular step 4. They broke it down into two smaller steps, and starting from the alternative semantics, they proved that removing indirections from the semantics does indeed yield an equivalent semantics [29]. But to this date, the remaining step resists rigorous proving, as Sánchez-Gil *et al.* report [28]. It seems that without blackholing and updates, the difference in heap evolution are too manyfold and intricate to be captured with a sufficiently elegant and handleable relation.

3.2 Denotational Blackholing

Considering the difficulties of following Launchbury's proof sketch directly, we revisited some of his steps. If it is so hard to relate the two operational semantics, maybe it works better to work just with the original operational semantics, and try to bridge the apparent mismatch between the operational semantics and the denotational semantics on the denotational side?

The first of the differences listed on page 11 – substitution vs. indirection – can be readily bridged by a substitution lemma ($[\![e]\!]_{\rho(y \mapsto \rho\ x)} = [\![e[y{:=}x]]\!]_\rho$), which is needed anyways in the correctness proof.

The third of the listed differences – whether updating is performed or not – is even easier to fix, as it does not affect the adequacy proof at all. In an adequacy proof we need to produce evidence for the *assumptions* of the corresponding natural semantics inference rule, which is then, in the last step, applied to produce the desired judgement. The removal of updates only changes the *conclusion* of the rule, so the adequacy proof is unchanged. There is an indirect effect, as in the adequacy proof we use the corresponding correctness result, and the correctness proof needs to address updates.

The remaining difference is the tricky one: how can we deal with blackholing in the adequacy proof? As during the evaluation of a (possibly recursive) variable binding the binding itself is removed from the heap, the denotation of the whole configuration changes. This is a problem, as there is no hope of proving $\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x \mapsto e, \Gamma\}\!\}} = \mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{\Gamma\}\!\}}$.

But what we could hope for is the following statement, which indicates that if a recursive expression has a denotation, then the expression has some (likely less defined) denotation even when all recursive calls are $\bot$:

$$\mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{x \mapsto e, \Gamma\}\!\}} n \neq \bot \implies \mathcal{N}[\![e]\!]_{\mathcal{N}\{\!\{\Gamma\}\!\}} n \neq \bot$$

This lemma follows from the following, also quite natural proposition, which expresses that if we allow the resourced semantics function to take at most $n+1$ steps, then the environment only matters in so far as at most n steps are passed to it:

$$\mathcal{N}[\![e]\!]_{\sigma}|_{(n+1)} = \mathcal{N}[\![e]\!]_{(\sigma|_n)}|_{(n+1)}$$

It turned out, however, that the resourced denotational semantics as introduced by Launchbury does not fulfill this property! By carefully capping the number of steps passed in the function equation for lambda expressions, we could fix this, and indeed prove adequacy of the resourced denotational semantics.

3.3 Gaining Assurance

These systems and proofs are, as the discussions in the previous sections showed, abundant with pitfalls. We would not believe our own proofs, had we not implemented all of them in Isabelle.

This includes a formalization of the similarity relation between the two denotational semantics from [27]. As it is a non-monotonous relation, we cannot simply define it as an inductive relation, but have to first define finite approximations and then manually take the limit. Once constructed, however, the details of the construction are no longer relevant – a blessing of the extensionality of Isabelle's logic.

Throughout our development we used the Isabelle implementation [32] of Nominal Logic [21] to elegantly deal with the contentious issues involving name binding. The interested reader can find full details in [6] and Breitner's award-winning thesis [5].

We have since extended our formalization [2] with boolean values and it has served as the target for a formal verification that a transformation performed by the Haskell compiler does not increase the number of allocations performed by the program [3, 4] – a property that can only be proven with a semantics as operational as Launchbury's.

4 Conclusion

Juvenal's question "Quis custodiet ipsos custodes?" can from a science-theoretic viewpoint be interpreted as an early precursor of later "constructivist" positions, which deny the possibility of objective knowledge and sound methodology. While we dismantled constructivist positions in the field of software technology [31, 30], we interpreted Juvenal's question as a challenge to produce soundness guarantees for various program analysis and software security techniques. Today we are satisfied to state our answer to Juvenal: "Illi Isabellistes Se Custodes Egregios Praestabant!".

References

1. Bischof, S., Breitner, J., Graf, J., Hecker, M., Mohr, M., Snelting, G.: Low-deterministic security for low-deterministic programs. Journal of Computer Security **26**, 335–336 (2018). DOI 10.3233/JCS-17984
2. Breitner, J.: The correctness of Launchbury's natural semantics for lazy evaluation. Archive of Formal Proofs (2013). URL http://afp.sf.net/entries/Launchbury.shtml
3. Breitner, J.: Formally proving a compiler transformation safe. In: Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015, pp. 35–46 (2015)
4. Breitner, J.: The safety of Call Arity. Archive of Formal Proofs (2015)
5. Breitner, J.: Lazy evaluation: From natural semantics to a machine-checked compiler transformation. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2016)
6. Breitner, J.: The adequacy of launchbury's natural semantics for lazy evaluation. J. Funct. Program. **28**, e1 (2018). DOI 10.1017/S0956796817000144. URL https://doi.org/10.1017/S0956796817000144
7. Breitner, J., Graf, J., Hecker, M., Mohr, M., Snelting, G.: On improvements of low-deterministic security. In: F. Piessens, L. Viganò (eds.) Proc. Principles of Security and Trust (POST), *Lecture Notes in Computer Science*, vol. 9635, pp. 68–88. Springer Berlin Heidelberg (2016)
8. Buiras, P., Russo, A.: Lazy programs leak secrets. In: NordSec, *Lecture Notes in Computer Science*, vol. 8208, pp. 116–122. Springer (2013)
9. Giffhorn, D.: Slicing of concurrent programs and its application to information flow control. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)
10. Giffhorn, D., Snelting, G.: A new algorithm for low-deterministic security. International Journal of Information Security **14**(3), 263–287 (2015)
11. Graf, J.: Information flow control with system dependence graphs — improving modularity, scalability and precision for object oriented languages. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2016)
12. Graf, J., Hecker, M., Mohr, M., Snelting, G.: Tool demonstration: JOANA. In: F. Piessens, L. Viganò (eds.) Proc. Principles of Security and Trust (POST), *Lecture Notes in Computer Science*, vol. 9635, pp. 89–93. Springer Berlin Heidelberg (2016)
13. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security **8**(6), 399–422 (2009)
14. Hölzl, J.: Construction and stochastic applications of measure spaces in higher-order logic. Dissertation, Technische Universität München, München (2013)

15. Küsters, R., Scapin, E., Truderung, T., Graf, J.: Extending and applying a framework for the cryptographic verification of Java programs. In: Proc. POST 2014, LNCS 8424, pp. 220–239. Springer (2014)
16. Launchbury, J.: A natural semantics for lazy evaluation. In: Principles of Programming Languages (POPL). ACM (1993). DOI 10.1145/158511.158618
17. Lochbihler, A.: Verifying a compiler for java threads. In: Proc. 19th European Symposium on Programming, ESOP 2010, *Lecture Notes in Computer Science*, vol. 6012, pp. 427–447 (2010)
18. Lochbihler, A.: A machine-checked, type-safe model of java concurrency : Language, virtual machine, memory model, and verified compiler. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)
19. Lochbihler, A.: Making the Java memory model safe. ACM Transactions on Programming Languages and Systems **35**(4), 12:1–12:65 (2014)
20. Peyton Jones, S.: Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. Journal of Functional Programming **2**(2), 127–202 (1992). DOI 10.1017/S0956796800000319
21. Pitts, A.M.: Nominal logic, a first order theory of names and binding. In: Theoretical Aspects of Computer Software (TACS) 2001, *Information and Computation*, vol. 186, pp. 165 – 193. Elsevier (2003). DOI 10.1016/S0890-5401(03)00138-X
22. Popescu, A., Hölzl, J., Nipkow, T.: Formal verification of language-based concurrent noninterference. J. Formalized Reasoning **6**(1), 1–30 (2013)
23. Popescu, A., Hölzl, J., Nipkow, T.: Formalizing probabilistic noninterference. In: Proc. Certified Programs and Proofs CPP, *Lecture Notes in Computer Science*, vol. 8307, pp. 259–275 (2013)
24. Popescu, A., Hölzl, J., Nipkow, T.: Noninterfering schedulers - when possibilistic noninterference implies probabilistic noninterference. In: Proc. Algebra and Coalgebra in Computer Science (CALCO), Lecture Notes in Computer Science, pp. 236–252 (2013)
25. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21**(1), 5–19 (2003)
26. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000, pp. 200–214 (2000)
27. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: Relating function spaces to resourced function spaces. In: Symposium on Applied Computing (SAC), pp. 1301–1308. ACM (2011). DOI 10.1145/1982185.1982469
28. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: Launchbury's semantics revisited: On the equivalence of context-heap semantics (work in progress). XIV Jornadas sobre Programación y Lenguajes pp. 203–217 (2014)
29. Sánchez-Gil, L., Hidalgo-Herrero, M., Ortega-Mallén, Y.: The role of indirections in lazy natural semantics. In: Perspectives of System Informatics (PSI) 2014, *LNCS*, vol. 8974. Springer (2015). DOI 10.1007/978-3-662-46823-4_24
30. Snelting, G.: Paul Feyerabend and software technology. International Journal on Software Tools for Technology Transfer **2**(1), 1–5 (1998)
31. Snelting, G.: Paul Feyerabend und die Softwaretechnologie. Informatik-Spektrum **21**(5), 273–276 (1998)
32. Urban, C., Kaliszyk, C.: General bindings and alpha-equivalence in nominal Isabelle. Logical Methods in Computer Science **8**(2) (2012). DOI 10.2168/LMCS-8(2:14)2012
33. Wasserrab, D.: From formal semantics to verified slicing – a modular framework with applications in language based security. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2010). URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020678
34. Wasserrab, D.: Information flow noninterference via slicing. Archive of Formal Proofs (2010)
35. Wasserrab, D., Lohner, D., Snelting, G.: On PDG-based noninterference and its modular proof. In: Proc. PLAS '09. ACM (2009). URL http://pp.info.uni-karlsruhe.de/uploads/publikationen/wasserrab09plas.pdf

36. Wasserrab, D., Nipkow, T., Snelting, G., Tip, F.: An operational semantics and type safety proof for multiple inheritance in C++. In: 21th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 345–362. ACM (2006)
37. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proc. CSFW, pp. 29–43. IEEE (2003)