Lecture Notes on Products and Sums

15-814: Types and Programming Languages
Jan Hoffmann

Lecture 7
Tuesday, September 15, 2026

1 Introduction

System T is very expressive if we consider definability of computable functions on the natural numbers. However, the only values we have are natural numbers and functions, and the type system prevents us from implementing the Church encodings of other data structures as we did in the λ-calculus.

In this lecture, we introduce *products* and *sums*, two basic kinds of data structures that should arguably be available in every programming language. Products are a *conjunctive combination* of data and sums are a *disjunctive combination* of data. Notably, sums are not available in many popular languages, which can be seen as the root cause of bugs like null-pointer dereference that arise when the type system is not tracking them.

You can think of the following presentation of products and sums as an extension of System T. However, the type and evaluation rules are modular and can equally be added to other languages that we will study. The rules do not rely on the fact that System T is a total language.

2 Products

Products are a conjunctive combination of data. Most popular programming languages feature a version of products. Examples of products are tuples, records, structs, and unit. The idea is to combine multiple data structures into a new data structure that contains all of them.

The most general version of products is finite (labeled) products, such as records, which are defined in [Har16]. In this lecture we present *pairs* and

unit, which are equally expressive but notationally simpler.

2.1 Static Semantics

Syntax The syntax of pairs and unit is defined as follows.

			Abstract	Concrete	
Typ	τ	::=	`unit`	1	unit type
			`prod`$(\tau_1;\tau_2)$	$\tau_1 \times \tau_2$	pair type
Exp	e	::=	`triv`	$\langle\rangle$	unit value
			`pair`$(e_1;e_2)$	$\langle e_1, e_2\rangle$	pair
			`proj`$[1](e)$	$e \cdot 1$	left projection
			`proj`$[2](e)$	$e \cdot 2$	right projection

The type `unit` is the type that has exactly one value, so the concrete syntax is often written as 1. The type `prod`$(\tau_1;\tau_2)$ is the type of pairs such that the first component is of type τ_1 and the second component is of type τ_2.

The expression `triv` is the introduction form for type `unit`. It is the only value of type `unit`. The expression `pair`$(e_1;e_2)$ is the introduction form for pairs. The elimination form for pairs is the projection `proj`$[i](e)$ for $i \in \{1,2\}$. It evaluates to the first ($i = 1$) or second ($i = 2$) component of the pair. There does not exist an elimination form for `unit` since the unit value is computationally uninteresting: We never have to inspect it because we know that a computation of type `unit` returns unit (or diverges if that is possible in the programming language we're working in).

Type Rules The type rules for pairs and unit are defined as follows.

$$\frac{}{\Gamma \vdash \texttt{triv} : \texttt{unit}}\ T_{\text{unit}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{pair}(e_1;e_2) : \tau_1 \times \tau_2}\ T_{\text{pair}}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \qquad i \in \{1,2\}}{\Gamma \vdash \texttt{proj}[i](e) : \tau_i}\ T_{\text{proj}}$$

The rule T_{unit} states that the unit value `triv` has type `unit` in every context. Rule T_{pair} states that if the type of the first component e_1 is τ_1 and the type of the second component e_2 is τ_2 then the type of the pair `pair`$(e_1;e_2)$

is $\tau_1 \times \tau_2$. Rule T_{proj} states that the projection to the ith component of an expression of type $\tau_1 \times \tau_2$ results in an expression of type τ_i if $i \in \{1, 2\}$. You can think of $i \in \{1, 2\}$ as a (regular) premise or (equivalently) as shorthand for defining two rules.

If we view the type rules as an extension of System T then we can show that the properties of the type system such as the substitution lemma and uniqueness of types still hold.

2.2 Dynamic Semantics

For *eager (or call-by-value) pairs*, values and transitions are defined as follows.

$$\frac{}{\texttt{triv}\ \mathsf{val}}\ V_{\text{unit}} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val}}{\texttt{pair}(e_1\texttt{;}\, e_2)\ \mathsf{val}}\ V_{\text{pair}}$$

$$\frac{e_1 \longmapsto e_1'}{\texttt{pair}(e_1\texttt{;}\, e_2) \longmapsto \texttt{pair}(e_1'\texttt{;}\, e_2)}\ E_{\text{pair1}}$$

$$\frac{e_1\ \mathsf{val} \qquad e_2 \longmapsto e_2'}{\texttt{pair}(e_1\texttt{;}\, e_2) \longmapsto \texttt{pair}(e_1\texttt{;}\, e_2')}\ E_{\text{pair2}}$$

$$\frac{e \longmapsto e'}{\texttt{proj}[i](e) \longmapsto \texttt{proj}[i](e')}\ E_{\text{proj1}} \qquad \frac{e_1\ \mathsf{val} \qquad e_2\ \mathsf{val} \qquad i \in \{1, 2\}}{\texttt{proj}[i](\texttt{pair}(e_1\texttt{;}\, e_2)) \longmapsto e_i}\ E_{\text{proj2}}$$

For an expression $\texttt{pair}(e_1\texttt{;}\, e_2)$, we first step inside e_1 using Rule E_{pair1}. When e_1 is a value, we use Rule E_{pair2} to step inside e_2. For an expression $\texttt{proj}[i](e)$, we first step inside e using Rule E_{proj1}. When e is a value of the form $\texttt{pair}(e_1\texttt{;}\, e_2)$ we step to e_i to retrieve the ith component.

For *lazy (or call-by-name)* evaluation of pairs, we remove the premises e_1 val and e_2 val from rule V_{pair}. We also remove rules E_{pair1} and E_{pair2} and the premises e_1 val and e_2 val from rule E_{proj2}.

2.3 Discussion

Type Soundness It is a great exercise to verify that progress and preservation still hold for System T extended with either eager or lazy products.

Records The most general form of tuples is *records*, also called *finite labeled products*. We have discussed pairs and very similar rules work for tuples. In addition, we could allow the user to specify the names of the components of a tuple instead of using natural numbers as in tuples. The resulting forms are often called records and are the most general form of products. You can find the rules for records in PFPL [Har16].

Function Arguments Consider again System T and observe that adding products immediately lets us define functions with multiple arguments and multiple return values. For example, $f : \texttt{nat} \times \texttt{nat} \to \texttt{nat} \times \texttt{nat}$ is a function that takes two natural numbers as arguments and returns two natural numbers. More precisely, functions still have one argument and one result but the argument or result can be a product.

So there is no need to explicitly generalize functions to have multiple arguments or results. Arguably, a generalization of functions that distinguishes a function with two arguments from a function with one argument that is a pair (like in Python) makes it harder to compose functions since you cannot directly apply a function with two arguments to a pair that is returned by another function.

3 Sums

Sums are the *disjunctive combination of data* and are dual to products. Examples of sums include Booleans, enumerations, option types, and void, the empty type. The idea of sums is to create a union of different data structures in which data is labeled based on the component of the union it is in.

The most general version of sums are finite (labeled) sums, which are available in functional languages such as SML, OCaml, and Haskell. In this lecture we discuss *binary sums* and *void*, which are equally expressive but notationally simpler.

3.1 Static Semantics

Syntax The syntax of binary sums and void is defined as follows.

			Abstract	Concrete	
Typ	τ	::=	`void`	0	void type
			$\texttt{sum}(\tau_1\texttt{;}\tau_2)$	$\tau_1 + \tau_2$	sum type
Exp	e	::=	$\texttt{inj}[1]\{\tau_1\texttt{;}\tau_2\}(e)$	$1 \cdot e$	left injection
			$\texttt{inj}[2]\{\tau_1\texttt{;}\tau_2\}(e)$	$2 \cdot e$	right injection
			$\texttt{case}(e\texttt{;}x_1.e_1\texttt{;}x_2.e_2)$	$\texttt{case}\ e\,\{\,x_1.e_1 \mid x_2.e_2\,\}$	case analysis
			$\texttt{absurd}\{\tau\}(e)$	$\texttt{absurd}(e)$	absurd

Void is the empty type, which is dual to unit in that it does not have an introduction form but the, somewhat mysterious, elimination form $\texttt{absurd}\{\tau\}(e)$ that we explain in the following. Binary sums have the two injections as introduction forms and the case analysis is the elimination form.

Type Rules The type rules for sums are defined as follows.

$$\frac{\Gamma \vdash e : \tau_i \qquad i \in \{1,2\}}{\Gamma \vdash \texttt{inj}[i]\{\tau_1\texttt{;}\tau_2\}(e) : \tau_1 + \tau_2}\ T_{\text{inj}}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{case}(e\texttt{;}x_1.e_1\texttt{;}x_2.e_2) : \tau}\ T_{\text{case}}$$

$$\frac{\Gamma \vdash e : \texttt{void}}{\Gamma \vdash \texttt{absurd}\{\tau\}(e) : \tau}\ T_{\text{absurd}}$$

In Rule T_{inj}, we ensure that e has type τ_i that is associated with the label i. In Rule T_{case}, the case analysis takes an expression e of type $\tau_1 + \tau_2$ and provides two branches: If e evaluates to the injection $\texttt{inj}[i]\{\tau_1\texttt{;}\tau_2\}(v)$ then v is bound to x_i in the evaluation of e_i. Therefore, the type of x_i is τ_i and the type of e_i is τ. Since we have the same type τ in both branches, the result type of the case analysis is also τ.

Rule T_{absurd} types the elimination form $\texttt{absurd}\{\tau\}(e)$ for values of type `void`. The expression e has type `void` and the type of $\texttt{absurd}\{\tau\}(e)$ is an arbitrary type τ, which the user selects with the annotation in the syntax. There are no values of type `void` since `void` has no introduction forms. But why do we then have an elimination form? First, because we can have it: Since there are no values of type `void`, the expression e does not evaluate

to a value. Therefore, it is safe to assign an arbitrary type τ to abort. Second, it matches the logical principle that anything (here τ) follows from *false* (here `void`). Third, there are situations in which it is actually beneficial in software engineering to express that some cases can be ignored because they cannot happen.[1]

3.2 Dynamic Semantics

The *eager (or call-by-value)* dynamics for void and binary sums is defined as follows.

$$\frac{e\ \mathsf{val} \qquad i \in \{1,2\}}{\mathtt{inj}[i]\{\tau_1;\tau_2\}(e)\ \mathsf{val}}\ V_{\mathsf{inj}}$$

$$\frac{e \longmapsto e' \qquad i \in \{1,2\}}{\mathtt{inj}[i]\{\tau_1;\tau_2\}(e) \longmapsto \mathtt{inj}[i]\{\tau_1;\tau_2\}(e')}\ E_{\mathsf{inj}}$$

$$\frac{e \longmapsto e'}{\mathtt{case}(e;x_1.e_1;x_2.e_2) \longmapsto \mathtt{case}(e';x_1.e_1;x_2.e_2)}\ E_{\mathsf{case1}}$$

$$\frac{e\ \mathsf{val} \qquad i \in \{1,2\}}{\mathtt{case}(\mathtt{inj}[i]\{\tau_1;\tau_2\}(e);x_1.e_1;x_2.e_2) \longmapsto [e/x_i]e_i}\ E_{\mathsf{case2}}$$

$$\frac{e \longmapsto e'}{\mathtt{absurd}\{\tau\}(e) \longmapsto \mathtt{absurd}\{\tau\}(e')}\ E_{\mathsf{absurd}}$$

An injection $\mathtt{inj}[i]\{\tau_1;\tau_2\}(e)\ \mathsf{val}$ is a value if the payload e is a value (Rule V_{inj}). There is no value of type `void`.

To evaluate an injection to a value, we repeatedly use Rule E_{inj} to step inside the payload.

To evaluate the case analysis $\mathtt{case}(e;x_1.e_1;x_2.e_2)$, we first step inside the argument e using Rule E_{case1}. If the argument of the case analysis has the form of a value $\mathtt{inj}[i]\{\tau_1;\tau_2\}(e)$ then we step to the respective case with the payload e substituted for the bound variable (Rule E_{case2}).

To evaluate $\mathtt{absurd}\{\tau\}(e)$ we use Rule E_{absurd} to step inside e using the premise $e \longmapsto e'$. Since e' cannot be a value, we can apply this rule over and

[1] I posted an example of such a situation on Ed.

over in an evaluation. This means that the program $\texttt{absurd}\{\tau\}(e)$ diverges, which can happen in a programming language with non-termination. Since System T is a total language, the rule E_{absurd} is never used in an evaluation.

For lazy (or call-by-name) sums, we drop the premise e val in Rule V_{inj}, the Rule E_{inj}, and the premise e val in Rule E_{case2}.

3.3 Examples and Discussion

Booleans The most prominent sum type are the Booleans, which we define as follows.

$$\begin{array}{rcl} \texttt{bool} & \triangleq & \texttt{unit} + \texttt{unit} \\ \texttt{true} & \triangleq & \texttt{inj}[1]\{\texttt{unit}\,;\,\texttt{unit}\}(\langle\rangle) \\ \texttt{false} & \triangleq & \texttt{inj}[2]\{\texttt{unit}\,;\,\texttt{unit}\}(\langle\rangle) \\ \texttt{if}(e; e_1; e_2) & \triangleq & \texttt{case}(e; x_1.e_1; x_2.e_2) \\ & & \text{where } x_1 \text{ and } x_2 \text{ are fresh} \end{array}$$

Options and Null Pointers Another well-known sum type is the option type $\texttt{opt}(\tau)$, which combines values of type τ with another value that indicates that the optional value of type τ is not available.

$$\begin{array}{rcl} \texttt{opt}(\tau) & \triangleq & \tau + \texttt{unit} \\ \texttt{null} & \triangleq & \texttt{inj}[2]\{\tau; \texttt{unit}\}(\langle\rangle) \\ \texttt{just}(e) & \triangleq & \texttt{inj}[1]\{\tau; \texttt{unit}\}(e) \\ \texttt{ifnull}(e; e_1; x_2.e_2) & \triangleq & \texttt{case}(e; x_1.e_1; x_2.e_2) \\ & & \text{where } x_1 \text{ is fresh} \end{array}$$

The use of option types eliminates the decades-old issue of *null-pointer dereference*, one of the most common bugs in software implemented in languages such as Java and C++. *Null-pointer dereference* is also the root cause of many security vulnerabilities. A null-pointer dereference occurs when you try to access a heap-allocated object that has not been created yet or is otherwise unavailable. The need to reference a not-yet-existing object naturally occurs when you create cyclic data structures such as doubly-linked lists.

Because of the prevalence of this bug, null-pointer dereference has been a focus area of research on static analysis and software engineering. However, even after effective mitigations had become available, null-pointer dereference still accounted for about 40% of memory bugs in Java projects [LTW+06] in 2006. In 2020, null-pointer dereference was still the most com-

mon logged error in Java production code[2]. This is in spite of the availability of effective detection tools and programming language mitigations such as a form of option types (using generics). Tony Hoare referred to null pointers as his "billion dollar mistake" because he introduced the concept in ALGOL W.[3]

Options eliminate the problem of null dereference because they combine null with the values of a type τ in a type-safe way. If we have a value e of type $\texttt{opt}(\tau)$ then the type system prevents us from using operations for τ on e. The only way to apply such operations is to use a case analysis $\texttt{ifnull}(e; e_1; x_2.e_2)$ and apply them to x_2 in the branch e_2. This forces us to safely handle the null case in the branch e_1. Moreover, this method is efficient because x_2 has the type τ in e_2. In this way, we know that we already performed the null check and do not have to repeat it in a function that is called with the argument x_2 inside e_2.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, page 25–33, New York, NY, USA, 2006. Association for Computing Machinery.

[2] https://www.harness.io/blog/10-exception-types-in-production-java-applications
[3] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Ho