Handout for part 1:

Higher-order Rewriting: motivation and definitions

1. Motivation

2

Common features from functional programming

$$
\begin{aligned}
&\texttt{map} :: (\alpha \Rightarrow \beta) \Rightarrow \mathsf{list}\ \alpha \Rightarrow \mathsf{list}\ \beta \\
&\texttt{map}\ F\ [] = [] \\
&\texttt{map}\ F\ (h\ ;\ t) = (F\ h)\ ;\ (\texttt{map}\ F\ t)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{double\_all} :: \mathsf{list\ int} \Rightarrow \mathsf{listint} \\
&\texttt{double\_all}\ lst = \texttt{map double}\ lst
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{handle} :: \mathsf{event} \Rightarrow \mathsf{list}\ (\mathsf{event} \Rightarrow \mathsf{result}) \Rightarrow \mathsf{list\ result} \\
&\texttt{handle}\ e\ callbacks = \texttt{map}\ (\lambda F.F\ e)\ callbacks
\end{aligned}
$$

Functional programming is a lot like term rewriting, but also has many structures that have no counterpart in traditional term rewriting systems. An important example of this is the ability to send *functions* as arguments to other functions, whether by sending an unapplied or partially applied function symbol, or constructing a lambda-expression.

This can both be used to express very pure, mathematical algorithms, but also to implement very practical solutions that for instance work with callbacks for event handlers.

3

Common features from functional programming

$$
\begin{aligned}
&\texttt{init} :: (\mathsf{int} \Rightarrow \alpha) \Rightarrow \mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{list}\ \alpha \\
&\texttt{init}\ F\ x\ y = \mathsf{if}\ (x \geq y)\ \mathsf{then}\ []\ \mathsf{else}\ ((F\ x)\ ;\ (\texttt{init}\ F\ (x+1)\ y))
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{filter} :: (\alpha \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{list}\ \alpha \Rightarrow \mathsf{list}\ \alpha \\
&\texttt{filter}\ F\ [] = [] \\
&\texttt{filter}\ F\ (h\ ;\ t) = \mathsf{if}\ (F\ h)\ \mathsf{then}\ (h\ ;\ (\texttt{filter}\ F\ t)) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{else}\ (\texttt{filter}\ F\ t)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{evens} :: \mathsf{int} \Rightarrow \mathsf{list\ int} \\
&\texttt{evens}\ n = \texttt{filter is\_even}\ (\texttt{init id}\ 0\ n)
\end{aligned}
$$

Typical examples often implement algorithms on integers or lists, but in larger programs all kinds of functions appear naturally.

Typical usage: expand $\mathtt{f}(s_1, \ldots, s_n)$ to $\lambda x_1 \ldots x_m.\mathtt{f}(s_1, \ldots, s_n, x_1, \ldots, x_m)$ if $\mathtt{f} :: \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$ with ι a base type. This is convenient when reasoning: we can assumpe that every occurrence of a function symbol $\mathtt{f}$ has the same number of arguments, which makes it easier to extend techniques from first-order term rewriting.

A downside of this approach is *polymorphism*. Consider for instance a rule like the following:

$\mathtt{head}(\mathtt{cons}(X, Y)) \to X \quad$ with $\quad \mathtt{head} :: \mathsf{list}(\alpha) \Rightarrow \alpha$

If we use η-expansion on all rules in a simply-typed system to maximally extend function symbols, then the instances of this rule will have different structures:

$$\begin{array}{rcl} \mathtt{head}_{\mathsf{int}}(\mathtt{cons}(X_{\mathsf{int}}, Y_{\mathsf{list(int)}})) & \to & X_{\mathsf{int}} \\ \mathtt{head}_{\mathsf{int}\Rightarrow\mathsf{bool}}(\mathtt{cons}(X_{\mathsf{int}\Rightarrow\mathsf{bool}}, Y_{\mathsf{list(int}\Rightarrow\mathsf{bool)}}), Z_{\mathsf{int}}) & \to & X_{\mathsf{int}\Rightarrow\mathsf{bool}} \cdot Z_{\mathsf{int}} \\ & \ldots & \end{array}$$

Here, we use a formalism that does not use η-expansion, but the techniques then apply to η-long variations of the HTRS, too.

27

Final exercise

Write a short HTRS (including signature!) that, given a list of natural numbers and a function mapping numbers to Booleans, finds the number of items in the list that satisfy the requirement.

4

Common features from functional programming

$$\begin{array}{l} \mathtt{fold} :: (\alpha \Rightarrow \beta \Rightarrow \beta) \Rightarrow \beta \Rightarrow \mathsf{list}\ \alpha \Rightarrow \beta \\ \mathtt{fold}\ F\ x\ [] = x \\ \mathtt{fold}\ F\ x\ (h\ ;\ t) = \mathtt{fold}\ F\ (F\ h\ x)\ t \\ \\ \mathtt{rec} :: (\mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int}) \Rightarrow \mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int} \\ \mathtt{rec}\ F\ x\ n = \mathsf{if}\ (n \leq 0)\ \mathsf{then}\ x\ \mathsf{else}\ (\mathtt{rec}\ F\ (F\ x\ n)\ (n-1)) \\ \\ \mathtt{exp} :: \mathsf{int} \Rightarrow \mathsf{int} \Rightarrow \mathsf{int} \\ \mathtt{exp}\ x\ y = \mathtt{rec}\ ((*)\ x)\ 1\ y \\ \\ \mathtt{greatest} :: \mathsf{list\ int} \Rightarrow \mathsf{int} \\ \mathtt{greatest}\ l = \mathtt{fold}\ (\lambda y, z.\mathsf{if}\ (z > y)\ \mathsf{then}\ z\ \mathsf{else}\ y)\ 0\ l \end{array}$$

Higher-order programming allows the programmer to isolate very basic common programming structures, such as "iterating over a list" or "iterating over numbers in a given range".

5

Shorthand using higher-order functions

$$\begin{array}{l} \mathtt{sumfun}\ F\ x = \mathsf{if}\ (x \leq 0)\ \mathsf{then}\ (F\ x) \\ \qquad\qquad\qquad \mathsf{else}\ ((F\ x) + (\mathtt{sumfun}\ F\ (x-1))) \\ \\ \mathtt{sumfun}\ F\ x = \mathtt{fold}\ (+)\ 0\ (\mathtt{map}\ F\ (\mathtt{init}\ \mathtt{id}\ 0\ x)) \\ \\ \mathtt{sumfun}\ F\ x = \mathtt{rec}\ (\lambda y, z.y + F(z))\ (F\ 0)\ x \end{array}$$

It is very common for functional programmers to use this ability to write highly compact code.

6

Higher-order functions appear naturally in...

- functional programming (as we just saw)
- object-oriented programming (we can view a class that implements a given interface as a collection of data and function symbols of specific types)
- mathematics (e.g., quantifiers) (mathematicians often reason about functions; for example the induction principle for natural numbers can be formulated as follows)

  $$\forall P :: \mathsf{nat} \Rightarrow \mathsf{bool}.(P(0) \wedge \forall x :: \mathsf{nat}.P(x) \Rightarrow P(x+1)) \Rightarrow \forall x :: \mathsf{nat}.P(x)$$

  (note that quantifiers are exactly higher-order functions that take a predicate as argument; the example of induction above is even third-order because the argument to $\forall$ is a function of type $(\mathsf{nat} \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$)

- theorem proving (in proof assistants like Coq, specifications of functions are provided in a form of higher-order term rewriting; aside from allowing us the ability to reason about mathematical structures like functions and predicates, higher-order specifications may even be useful for reasoning about naturally first-order definitions, since higher-order specifications allow for a kind of modularity that makes it easier to prove termination requirements and helps to analyse parts of a definition in isolation)

$$\begin{array}{rcl} \mathtt{ack}\ \mathtt{0}\ n & \to & \mathtt{s}\ n \\ \mathtt{ack}\ (\mathtt{s}\ m)\ \mathtt{0} & \to & \mathtt{ack}\ m\ (\mathtt{s}\ \mathtt{0}) \\ \mathtt{ack}\ (\mathtt{s}\ m)\ (\mathtt{s}\ n) & \to & \mathtt{ack}\ m\ (\mathtt{ack}\ (\mathtt{s}\ m)\ n) \end{array}$$

$\Longrightarrow$

$$\begin{array}{rcl} \mathtt{helper}\ F\ \mathtt{0} & \to & F\ (\mathtt{s}\ \mathtt{0}) \\ \mathtt{helper}\ F\ (\mathtt{s}\ n) & \to & F\ (\mathtt{helper}\ F\ n) \\ \mathtt{ack}\ \mathtt{0} & \to & \mathtt{s} \\ \mathtt{ack}\ (\mathtt{s}\ m) & \to & \mathtt{helper}\ (\mathtt{ack}\ m) \end{array}$$

7

A first idea

Question: do we need more than we already have?

Idea:

- application symbol @ with arity 2
- all other symbols: arity 0

Examples:

$$\begin{array}{rcl} @(@(\mathtt{map}, F), []) & \to & [] \\ @(@(\mathtt{map}, F), @(@(\mathtt{cons}, h), t)) & \to & @(@(\mathtt{cons}, @(F, h)), @(@(\mathtt{map}, F), t)) \\ @(@(@(\mathtt{rec}, F), x), \mathtt{0}) & \to & x \\ @(@(@(\mathtt{rec}, F), x), @(\mathtt{s}\ y)) & \to & @(@(@(\mathtt{rec}, F), \\ & & \quad @(@(F, x), @(\mathtt{s}, y))), \\ & & \quad y) \end{array}$$

This is of course not particularly readable, but we can use some syntactic sugar: we present the function symbol @ as an *infix* symbol $\cdot$, and say that $\cdot$ is left-associative; i.e., $a \cdot b \cdot c = @(@(a, b), c)$. Then we end up with the much more pleasant specification:

$$\begin{array}{rcl} \mathtt{map} \cdot F \cdot [] & \to & [] \\ \mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot h \cdot t) & \to & \mathtt{cons} \cdot (F \cdot h) \cdot (\mathtt{map} \cdot F \cdot t) \\ \mathtt{rec} \cdot F \cdot x \cdot \mathtt{0} & \to & x \\ \mathtt{rec} \cdot F \cdot x \cdot (\mathtt{s} \cdot y) & \to & \mathtt{rec} \cdot F \cdot (F \cdot x \cdot (\mathtt{s} \cdot y)) \cdot y \end{array}$$

8

Applicative rewriting

Question: what about lambda?

$$\mathtt{handle} \cdot e \cdot \mathit{callbacks} \ \to \ \mathtt{map} \cdot \text{“}(\lambda F.F\ e)\text{”} \cdot \mathit{callbacks}$$

Hence, for simplicity in this course I will limit interest to simply-typed HTRSs, and we may discuss how the results extend to their polymophically-typed counterparts.

25

More higher-order definitions

I here present one definition of higher-order term rewriting, but if you meet someone else in the field, they would likely give you a different one. This is because there have been many different definitions, for a variety of purposes. To name some examples...

- CS [Aczel, 1978]
- CRS [Klop, 1980]
- ERS [Khasidashvili, 1990]
- HRS [Nipkow, 1991]
- AFS [Jouannaud, Okada, 1991]
- HORS [van Oostrom, 1994]
- IDTS [Blanqui, 2000]
- STRS [Kusakari, 2001]
- STTRS [Yamada, 2001]
- Nominal rewriting [Gabbay, Pitts, 2002]
- AFSM [Kop, 2012]
- ...

On top of that, there are many variations and restrictions, for example limiting the output types of rules, the shapes of the left-hand sides or the kinds of function symbols that are included. Nor does it account for different type systems, such as (various kinds of) polymorphism or dependent types.

Some of these systems are fundamentally different; others just seem to be small variations of the same idea – but small syntactic differences can actually make a significant difference for analysis!

26

Eta-expansion

An example of a feature that varies in different styles of higher-order term rewriting is whether or not equality is considered modulo η

$$s =_\eta \lambda x_\sigma.(s \cdot x_\sigma)$$

if $s :: \sigma \Rightarrow \tau$ and x_σ does not occur in s, and s not an abstraction

When reasoning about mathematics, it is natural to consider two functions equal if and only if their value on all input is the same. In such a setting, reasoning modulo η makes a lot of sense. When considering more syntactic applications, like functional programming, this is less natural.

4. Discussion

23

The limitation of simple types

Question: what if I want a list of numbers **and** a list of booleans **and** a list of functions?

What should be the type of `cons`?

Solution: have separate types intlist, boollist, inttoboollist, ...

$\Longrightarrow$ also make copies of `map`, `filter`, `fold`, etc. for each

Alternative solution: use wrappers

$$\begin{array}{rcl} \texttt{cons} & :: & \mathsf{A} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list} \\ \texttt{wrapint} & :: & \mathsf{int} \Rightarrow \mathsf{A} \\ \texttt{wrapbool} & :: & \mathsf{bool} \Rightarrow \mathsf{A} \end{array}$$

However, this is also not ideal: it requires a lot of encoding, and using wrappers like this throughout the HTRS loses many of the benefits that having types brings.

24

Shallow polymorphism

Yet, functional programming languages don't have this problem! This is because they do not use simple types, but instead use *polymorphic* types. Why not take a leaf out of their pocket?

Preferred type: $\texttt{map} :: (\alpha \Rightarrow \beta) \Rightarrow \mathsf{list}(\alpha) \Rightarrow \mathsf{list}(\beta)$

Idea: polymorphically-typed rules correspond to a set of simply-typed rules!

That is, including a symbol like `map` above in the signature, could be viewed as including the infinitely many function symbols $\texttt{map}_{\sigma,\tau} :: (\sigma \Rightarrow \tau) \Rightarrow$ "$\mathsf{list}(\sigma)$" $\Rightarrow$ "$\mathsf{list}(\tau)$", where $\mathsf{list}(\sigma)$ is a new base type for any type-variable-free type σ. And the usual map rule,

$$\texttt{map}(F, \texttt{cons}(h, t)) \to \texttt{cons}(F \cdot h, \texttt{map}(F, t))$$

would then correspond to the infinite set of rules:

$$\begin{array}{l} \texttt{map}_{\sigma,\tau}(F_{\sigma\Rightarrow\tau}, \texttt{cons}_{\sigma}(h_{\sigma}, t_{\mathsf{list}(\tau)})) \to \\ \qquad \texttt{cons}_{\tau}(F_{\sigma\Rightarrow\tau} \cdot h_{\sigma}, \texttt{map}_{\sigma,\tau}(F_{\sigma\Rightarrow\tau}, t_{\mathsf{list}(\tau)})) \end{array}$$

Thus, results on simply-typed HTRSs can be transferred to polymorphically-typed ones: to show that a set of rules satisfies a certain property, we then need to show that the property holds for all the simply-typed *instances* of these rules.

A simple solution could be to encode a λ-expression as a fresh function symbol:

$$\begin{array}{rcl} \texttt{handle} \cdot e \cdot \mathit{callbacks} & \to & \texttt{map} \cdot (\texttt{helper} \cdot e) \cdot \mathit{callbacks} \\ \texttt{helper} \cdot e \cdot F & \to & F \cdot e \end{array}$$

Downsides

- λ-expressions *in the input* cannot be represented (but: may not be needed!)
  If we want to for instance prove termination of *all* terms, we would have to include rules for all possible λ-terms into the TRS to analyse this. This creates an infinite system, which is not typically great for analysis. But: in practical applications we are often concerned with *specific* start terms, and it may be possible to exclude arbitrary lambda-terms from this.

- Having only one root symbol makes analysis harder
  Methods like the recursive path ordering tend to be very weak on large systems where all rules have the same root symbol: we cannot even prove something like $\texttt{f} \cdot x >_{\texttt{rpo}} \texttt{g} \cdot x \cdot x$. So, we may still need to develop dedicated analysis methods.

9

Exercise: playing with applicative systems

1. $$\begin{array}{rcl} \texttt{rec} \cdot F \cdot x \cdot \texttt{0} & \to & x \\ \texttt{rec} \cdot F \cdot x \cdot (\texttt{s} \cdot y) & \to & \texttt{rec} \cdot F \cdot (F \cdot x \cdot (\texttt{s} \cdot y)) \cdot y \end{array}$$
   Use `rec` to define **addition**, **multiplication** and **factorial**. You are allowed to use lambda (or define helper functions), but don't write your own recursive functions.

2. Prove termination of
   $$\texttt{f} \cdot x \to \texttt{g} \cdot x \cdot x$$
   (There are no additional function symbols – so also no lambda)

2. Challenges

10

Termination of map

Recall:

$$\begin{array}{rcl} \mathtt{map} \cdot F \cdot [] & \to & [] \\ \mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot h \cdot t) & \to & \mathtt{cons} \cdot (F \cdot h) \cdot (\mathtt{map} \cdot F \cdot t) \end{array}$$

Debate: is this TRS terminating?

Debate: what if we add the following rules?

$$\begin{array}{rcl} \mathtt{g} \cdot x \cdot F & \to & F \cdot x \\ \mathtt{tmp} \cdot x & \to & \mathtt{map} \cdot (\mathtt{g} \cdot x) \cdot x \end{array}$$

These additions may **seem** quite innocuous: they are just the encodings of (typable) lambda terms!

11

Termination of map

$$\begin{array}{rcl} \mathtt{map} \cdot F \cdot [] & \to & [] \\ \mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot h \cdot t) & \to & \mathtt{cons} \cdot (F \cdot h) \cdot (\mathtt{map} \cdot F \cdot t) \\ \mathtt{g} \cdot x \cdot F & \to & F \cdot x \\ \mathtt{tmp} \cdot x & \to & \mathtt{map} \cdot (\mathtt{g} \cdot x) \cdot x \end{array}$$

Answer: **NO**

Consider: Let $\omega := \mathtt{cons} \cdot \mathtt{tmp} \cdot []$

$$\begin{array}{rcl} & & \mathtt{map} \cdot (\mathtt{g} \cdot \omega) \cdot \omega \\ & = & \mathtt{map} \cdot \underbrace{(\mathtt{g} \cdot \omega)}_{F} \cdot (\mathtt{cons} \cdot \mathtt{tmp} \cdot \underbrace{[]}_{h} \ \underbrace{)}_{t} \\ & \to & \mathtt{cons} \cdot (\mathtt{g} \cdot \omega \cdot \mathtt{tmp}) \cdot (\mathtt{map} \cdot (\mathtt{g} \cdot \omega) \cdot []) \\ & = & \mathtt{cons} \cdot (\mathtt{g} \cdot \omega \cdot \mathtt{tmp}) \cdot [\cdots] \\ & \to & \mathtt{cons} \cdot (\mathtt{tmp} \cdot \omega) \cdot [\cdots] \\ & \to & \mathtt{cons} \cdot (\mathtt{map} \cdot (\mathtt{g} \cdot \omega) \cdot \omega) \cdot [\cdots] \end{array}$$

So $\mathtt{map} \cdot (\mathtt{g} \cdot \omega) \cdot \omega$ reduces to a term that has $\mathtt{map} \cdot (\mathtt{g} \cdot \omega) \cdot \omega$ as a subterm, proving non-termination.

12

Problem: termination of map

Discussion: is there an inherent problem that makes systems like **this** non-terminating?

$$\begin{array}{rcl} \mathtt{map} \cdot F \cdot [] & \to & [] \\ \mathtt{map} \cdot F \cdot (\mathtt{cons} \cdot h \cdot t) & \to & \mathtt{cons} \cdot (F \cdot h) \cdot (\mathtt{map} \cdot F \cdot t) \\ \mathtt{g} \cdot x \cdot F & \to & F \cdot x \\ \mathtt{tmp} \cdot x & \to & \mathtt{map} \cdot (\mathtt{g} \cdot x) \cdot x \end{array}$$

21

Exercise

Assign types to all symbols in the following rules. Use $\mathtt{cons} : \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$ and $\mathtt{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$.

$$\begin{array}{rcl} \mathtt{filter}(F, []) & \to & [] \\ \mathtt{filter}(F, \mathtt{cons}(h, t)) & \to & \mathtt{test}(F \cdot h, , F, h, t) \\ \mathtt{test}(\mathtt{true}, F, h, t) & \to & \mathtt{cons}(h, \mathtt{filter}(F, t)) \\ \mathtt{test}(\mathtt{false}, F, h, t) & \to & \mathtt{filter}(F, t) \end{array}$$

$$\begin{array}{rcl} \mathtt{I}(\mathtt{0}) & \to & \mathtt{0} \\ \mathtt{I}(\mathtt{s}(x)) & \to & \mathtt{s}(\mathtt{twice}(\mathtt{I}, x)) \\ \mathtt{twice}(F) & \to & \lambda x.F \cdot F \cdot x \end{array}$$

22

For those who like a bigger challenge!

Assign types to all symbols in the following rules. Use $\mathtt{s} : \mathsf{ord} \Rightarrow \mathsf{ord}$, and let the output type of rec be A.

$$\begin{array}{rcl} \mathtt{rec}(0, Y, F, G) & \to & Y \\ \mathtt{rec}(\mathtt{s}(X), Y, F, G) & \to & F \cdot X \cdot \mathtt{rec}(X, Y, F, G) \\ \mathtt{rec}(\mathtt{lim}(H), Y, F, G) & \to & G \cdot H \cdot (\lambda x_{\mathsf{nat}}.\mathtt{rec}(H \cdot x_{\mathsf{nat}}, Y, F, G)) \end{array}$$

- $C[\ell\gamma] \to_{\mathcal{R}} C[r\gamma]$ if $\ell \to r$ is a rule
- $C[\ (\lambda x_\sigma.s) \cdot t\ ] \to_{\mathcal{R}} C[\ s[x := t]\ ]$ (β-reduction)

Notation:

- we typically omit variable types when clear from context (we usually make sure that every occurrence of a variable in a given term/rule carries the same type)
- use uncurried notation $\mathtt{f}(s_1, \ldots, s_n)$ for $\mathtt{f} \cdot s_1 \cdots s_n$ (but we still allow partially applied terms to exist, e.g., $\mathtt{add}$, $\mathtt{add}(\mathtt{0})$ and $\mathtt{add}(\mathtt{s}(x), \mathtt{0})$ are all distinct terms)

19

Example HTRS

$$\begin{array}{rcl} \mathtt{map}(F, []) & \to & [] \\ \mathtt{map}(F, \mathtt{cons}(x, y)) & \to & \mathtt{cons}(F \cdot x, \mathtt{map}(F, y)) \\ \mathtt{double}(x) & \to & \mathtt{map}(\lambda y.\mathtt{add}(y, x)) \end{array}$$

Signature:

$$\begin{array}{lcl} \mathtt{add} & :: & \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat} \\ \mathtt{double} & :: & \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list} \\ [] & :: & \mathsf{list} \\ \mathtt{cons} & :: & \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list} \\ \mathtt{map} & :: & (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{list} \Rightarrow \mathsf{list} \end{array}$$

Note that the last rule has a type $\mathsf{list} \Rightarrow \mathsf{list}$ – this is perfectly allowed! Note also that y is used with different types in the last two rules. This is also allowed by our notational rules – we only have to be consistent with variable naming in the same rule. (But in case of confusion, feel free to add the type subscript or to use a fixed type for each name when writing your own examples.)

Note also that this presentation is syntactic sugar for:

$$\begin{array}{rcl} \mathtt{map} \cdot F_{\mathsf{nat}\Rightarrow\mathsf{nat}} \cdot [] & \to & [] \\ \mathtt{map} \cdot F_{\mathsf{nat}\Rightarrow\mathsf{nat}} \cdot (\mathtt{cons} \cdot x_{\mathsf{nat}} \cdot y_{\mathsf{list}}) & \to & \mathtt{cons} \cdot (F_{\mathsf{nat}\Rightarrow\mathsf{nat}} \cdot x_{\mathsf{nat}}) \cdot (\mathtt{map} \cdot F_{\mathsf{nat}\Rightarrow\mathsf{nat}} \cdot y_{\mathsf{list}}) \\ \mathtt{double} \cdot x_{\mathsf{nat}} & \to & \mathtt{map} \cdot (\lambda y_{\mathsf{nat}}.\mathtt{add} \cdot y_{\mathsf{nat}} \cdot x_{\mathsf{nat}}) \end{array}$$

20

Group exercise

Challenge: find the type of $\mathtt{rec}$!

$$\begin{array}{rcl} \mathtt{rec}(\mathtt{0}, Y, F) & \to & Y \\ \mathtt{rec}(\mathtt{s}(X), Y, F) & \to & F \cdot X \cdot \mathtt{rec}(X, Y, F) \end{array}$$

Signature:

$$\begin{array}{lcl} \mathtt{0} & :: & \mathsf{nat} \\ \mathtt{s} & :: & \mathsf{nat} \Rightarrow \mathsf{nat} \\ \mathtt{rec} & :: & \mathsf{nat} \Rightarrow A \Rightarrow (\mathsf{nat} \Rightarrow A \Rightarrow A) \Rightarrow A \end{array}$$

Idea: λ-calculus is non-terminating, so obviously we should not allow (arbitrary) λ-terms!

(But: how to design termination methods that would reject $\mathtt{g} \cdot x \cdot F \to F \cdot x$ or $\mathtt{tmp} \cdot x \to \mathtt{map} \cdot (\mathtt{g} \cdot x) \cdot x$?)

Idea: the problem arises due to **duplication**; can we avoid that?

(Not if we want functions like $\mathtt{map}$ and $\mathtt{fold}\ldots$)

Idea: the problem arises due to the construction of an untypable term $\mathtt{map} \cdot (\mathtt{g} \cdot \omega) \cdot \omega$

(Written as, for instance, an OCaml program, the rules above are terminating! So can we just limit interest to typable terms?)

13

Types: avoiding undesirable terms

Without type restrictions you can build terms such as:

$$\mathtt{add}(\mathtt{0}, \mathtt{apple})$$

This is a fully first-order example that illustrates the issue: when analysing term rewriting systems without types, we are also analysing a lot of trash: terms that have no meaning or purpose, but that exist as a side effect of the definition.

In higher-order rewiting this problem is exacerbated, because you end up conflating data and functions:

$$\mathtt{map}(\mathtt{map}(F, []), \mathtt{s}(\mathtt{pear})) \to \mathtt{map}([], \mathtt{s}(\mathtt{pear}))$$

14

Problems of type-insensitive analysis

$$\begin{array}{rcl} \mathtt{f}(\mathtt{0}) & \to & \mathtt{0} \\ \mathtt{f}(\mathtt{s}(x)) & \to & \mathtt{f}(x) \\ \mathtt{cost}(\mathtt{apple}) & \to & \mathtt{s}(\mathtt{0}) \\ \mathtt{cost}(\mathtt{pear}) & \to & \mathtt{s}(\mathtt{s}(\mathtt{0})) \end{array}$$

Question: are the following properties satisfied?

- $\mathtt{f}(t)$ reduces to $\mathtt{0}$ for all t
- every ground term reduces to a constructor normal form

This is very relevant for analysis methods like rewriting induction, which are in principle defined for first-order rewriting!

And even in a fully first-order system, typing matters for termination:

$$\begin{array}{rcl} \mathtt{f}(x, \mathtt{1}, \mathtt{2}) & \to & \mathtt{f}(x, x, x) \\ \mathtt{chooselist}(x, y) & \to & x \\ \mathtt{chooselist}(x, y) & \to & y \end{array}$$

This is only non-terminating if we are allowed to construct a term $\mathtt{chooselist}(\mathtt{1}, \mathtt{2})$!

15

Typing a term rewriting system (intuition)

Idea: add types to function symbols

Example: $0 :: \mathsf{nat}$, $\mathtt{apple} :: \mathsf{fruit}$, $\mathtt{banana} :: \mathsf{fruit}$, $\mathtt{s} :: \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathtt{cost} :: \mathsf{fruit} \Rightarrow \mathsf{nat}$, $\mathtt{add} :: \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, $[] :: \mathsf{list}$, $\mathtt{cons} :: \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$

Requirement: terms must be well-typed!

- **Terms:** $\mathtt{s}(\mathtt{add}(0, 0))$ and $\mathtt{cons}(0, \mathtt{cons}(\mathtt{add}(0, \mathtt{cost}(\mathtt{apple})), []))$
- **Not terms:** $\mathtt{cons}(0, \mathtt{banana})$ and $\mathtt{s}([])$

Variables: carry an *implicit* type. They must be typed consistently within the same term or rule.

- **Allowed:** $\mathtt{add}(x, x)$
- **Not allowed:** $\mathtt{cons}(x, x)$

Reduction: unchanged! (But: rules must be type-preserving; that is, the left- and right-hand side must have the sametype.)

3. Definition

16

Simple types

Fix: a set of **base types**

For example: nat, int, bool, fruit, list

Types are:

- all base types
- if σ and τ are types, then $\sigma \Rightarrow \tau$

Notation: the type arrow is **right-associative**

So $\sigma \Rightarrow \tau \Rightarrow \rho$ is just $\sigma \Rightarrow (\tau \Rightarrow \rho)$

And: all types can be written as $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota$ with ι a base type

17

HTRSs

We are now ready to formally define a higher-order term rewriting system.

Signature: we assume given a set $\mathcal{F}$ of pairs $\mathtt{f} :: \sigma$ with $\mathtt{f}$ a function symbol and σ a simple type

Example: $\mathcal{F} = \{0 :: \mathsf{nat}, \mathtt{s} :: \mathsf{nat} \Rightarrow \mathsf{nat}, \mathtt{add} :: \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}\}$

Variables: we assume given a set $\mathcal{V}$ of variables

Terms: are all expressions s such that $s :: \sigma$ can be deduced for some simple type σ using the rules:

- If $(\mathtt{f} :: \sigma) \in \mathcal{F}$ then $\mathtt{f} :: \sigma$
- If $x \in \mathcal{V}$ and σ a simple type then $x_\sigma :: \sigma$
- If $s :: \sigma \Rightarrow \tau$ and $t :: \sigma$ then $s \cdot t :: \tau$
- if $s :: \tau$, $x \in \mathcal{V}$ and σ a simple type, then $\lambda x_\sigma.s :: \sigma \Rightarrow \tau$

$\Longrightarrow$ the application operator is not a function symbol!

α-renaming: term equality is modulo renaming of bound variables (both the binder and all its occurrences in the subterm below the λ); for example, we consider $\lambda x_{\mathsf{nat}}.(\mathtt{s} \cdot x_{\mathsf{nat}})$ equal to $\lambda y_{\mathsf{nat}}.(\mathtt{s} \cdot y_{\mathsf{nat}})$. Note that we may change the *name*, but not the *type*. We can only change names of bound variables if this does not cause free variables to be captured; e.g., $\lambda x_{\mathsf{nat}}.\mathtt{add} \cdot x_{\mathsf{nat}} \cdot y_{\mathsf{nat}}$ is not the same as $\lambda y_{\mathsf{nat}}.\mathtt{add} \cdot y_{\mathsf{nat}} \cdot y_{\mathsf{nat}}$!

18

HTRSs (continued)

Rules: pairs of terms $\ell \to r$ **with the same type**

Reduction: