

Refinement Calculus

Foundations and Applications

Ralph-Johan Back and Joakim von Wright

Åbo Akademi University and
Turku Centre for Computer Science

I. Reasoning in Hierarchies

- A. A framework for hierarchical reasoning
- B. Predicate transformers – a program hierarchy
- C. Reasoning about programs and loop structures

II. Reasoning with Contracts

- A. Basic theory of contracts
- B. Application: UML use cases
- C. Extension: temporal reasoning with contracts

Motivation: Refining a Program

- Given a small program over integer variables

do $x > 0 \rightarrow \langle y := y + x \rangle ; [x := x' \mid 0 \leq x' < x]$ od

Motivation: Refining a Program

- Given a small program over integer variables

do $x > 0 \rightarrow \underline{\langle y := y + x \rangle ; [x := x' \mid 0 \leq x' < x]}$ od

- Zoom on the body of the loop

Motivation: Refining a Program

- Given a small program over integer variables

do $x > 0 \rightarrow \langle y := y + x \rangle ; \underline{[x := x' \mid 0 \leq x' < x]}$ od

- Zoom on the body of the loop
 on the second statement of the sequence

Motivation: Refining a Program

- Given a small program over integer variables

do $x > 0 \rightarrow \langle y := y + x \rangle ; [x := x' \mid \underline{0 \leq x' < x}]$ od

- Zoom on the body of the loop
on the second statement of the sequence
on the right of the assignment

Motivation: Refining a Program

- Given a small program over integer variables

$\text{do } x > 0 \rightarrow \langle y := y + x \rangle ; [x := x' \mid \underline{0 \leq x' < x}] \text{ od}$

- Zoom on the body of the loop
on the second statement of the sequence
on the right of the assignment statement

- Make deterministic, using the “fact” that $x > 0$

$\text{do } x > 0 \rightarrow \langle y := y + x \rangle ; \langle x := x - 1 \rangle \text{ od}$

Hierarchical Reasoning

- What is the formal basis for such reasoning?
 - higher-order logic
 - window inference
- How can it be described and presented?
 - structured derivations
- How can it be used in practice?
 - in ordinary mathematics
 - in program reasoning

I. Reasoning in Hierarchies

- A. A framework for hierarchical reasoning
- B. Predicate transformers – a program hierarchy
- C. Reasoning about programs and loop structures

Higher-Order Logic

- Like a typed predicate calculus, with higher-order quantification
- Based on Church's simple theory of types
- Used for mechanised reasoning: HOL, PVS, Isabelle
- Allows formulas like

$$(\forall P \bullet P.0 \wedge (\forall n \bullet P.n \Rightarrow P.(n+1))) \Rightarrow (\forall n \bullet P.n)$$

- Here: one possible formulation of higher-order logic

Types and Terms

- Type syntax: $\tau ::= \alpha \mid op^n(\tau_1, \dots, \tau_n)$
 - primitive type operators: Ind, Bool, \rightarrow
 - new types are defined using a principle of type definition
- Term syntax: $t ::= c \mid v \mid t.t' \mid (\lambda v \bullet t)$
 - primitive constants: \Rightarrow $=$ ϵ
 $\Rightarrow: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}$ $=: \alpha \rightarrow \alpha \rightarrow \text{Bool}$ $\epsilon: (\alpha \rightarrow \text{Bool}) \rightarrow \alpha$
 - ϵ is Hilbert's choice, $\epsilon.t$ returns some element of the "set" t
 - new constants can be added (conservatively) by definition $c = t$

Writing Terms

- Type inference is decidable
 - we do not have to write out types
- Syntactic sugaring:
 - infix constants: write $t = t'$ for $= .t.t'$
 - binders: write $(\forall v \bullet t)$ for $\forall.(\lambda v \bullet t)$
- Example: $(\forall P \bullet P.0 \wedge (\forall n \bullet P.n \Rightarrow P.(n+1))) \Rightarrow (\forall n \bullet P.n)$

Axioms and Inference Rules

- Sequents have form $\Gamma \vdash t$ where Γ is a set of terms
- Theorem is sequent proved from axioms and definitions using inference rules
- Five axioms, for example
 - boolean cases: $\vdash (\forall b : \text{Bool} \bullet b = \text{F} \vee b = \text{T})$
 - truth: $\vdash \text{T}$
- Eight inference rules, for example
 - Abstraction:
$$\frac{\Gamma \vdash t = t'}{\Gamma \vdash (\lambda v \bullet t) = (\lambda v \bullet t')}$$
 - Modus Ponens:
$$\frac{\Gamma_1 \vdash t_1 \quad \Gamma_2 \vdash t_1 \Rightarrow t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

Logic of Functions

- Abstraction $(\lambda v \cdot t)$ models function
- Application $t.t'$ models application of function to argument
- Conversion rules for handling functions, e.g.,
 - alpha conversion: $(\lambda v \cdot t) = (\lambda u \cdot t[v := u])$ if u nfi t
 - beta conversion: $(\lambda v \cdot t).t' = t[v := t']$ if ...
- Define function composition: $;$ $= (\lambda f \cdot \lambda g \cdot \lambda x \cdot g.(f.x))$
 - with syntactic sugaring: $(f ; g).x = g.(f.x)$

Logic of Booleans

- We are interested in proofs that transform a term
 - tautology: prove $t \equiv t'$
 - verification: prove $t \Leftarrow T$
 - contradiction: prove $t \Rightarrow F$
- General format for terms in sequents is thus $t R t'$ where R is
 - reflexive: $t R t$
 - transitive: from $t R t'$ and $t' R t''$ deduce $t R t''$
- Example \forall -elimination:
$$\frac{\vdash s \Rightarrow t}{\vdash s \Rightarrow (\forall v \cdot t)} \text{ if } v \text{ nfi } s$$

Generalising to Lattice Types

- Boolean \Rightarrow , \equiv , \Leftarrow are special cases of lattice \sqsubseteq , $=$, \sqsupseteq
- Most inference rules for booleans can be generalised to lattices
 - exception: rules for “two-valued” and “truth”
- Example: \sqcap -introduction
$$\frac{\Gamma \vdash s \sqsubseteq t \quad \Gamma' \vdash s \sqsubseteq t'}{\Gamma \cup \Gamma' \vdash s \sqsubseteq t \sqcap t'}$$
- Rules for booleans, sets, relations, are special cases

Window Inference

- Background: contextual rewriting
 - rewrite a subterm using information collected while traversing the syntax tree
- As logical method for proof in mathematics: Robinson and Staples
- Generalised to preorders and mechanised: Grundy
- The Refinement Calculator: mechanised support with GUI built on HOL

Window Inference: An Example

- The aim is to transform $(x + y = 3) \wedge (y = x - 1)$ into something
 - equivalent
 - simpler
- Window inference works with the following components:
 - a relation to be preserved
 - a context: facts that can be assumed
 - a focus: the term that is to be transformed
 - a stack of theorems: what has been proved so far

Window Inference: An Example

- Relation: \equiv
- Context: \emptyset
- Focus: $(x + y = 3) \wedge (y = x - 1)$
- Action:
- Theorems: $(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$

Window Inference: An Example

- Relation: \equiv
- Context: \emptyset
- Focus: $(x + y = 3) \wedge (y = x - 1)$

- Action: zoom in on left conjunct

- Theorems: $(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$

Window Inference: An Example

- Relation: \equiv

- Context: $y = x - 1$

- Focus: $x + y = 3$

- Action:

- Theorems: $x + y = 3 \equiv x + y = 3$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $y = x - 1$
- Focus: $x + y = 3$

- Action: rewrite using context information

- Theorems: $x + y = 3 \equiv x + y = 3$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv

- Context: $y = x - 1$

- Focus: $x + (x - 1) = 3$

- Action:

- Theorems: $x + y = 3 \equiv x + (x - 1) = 3$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $y = x - 1$
- Focus: $x + (x - 1) = 3$
- Action: rewrite using standard equation solving rules

- Theorems: $x + y = 3 \equiv x + (x - 1) = 3$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $y = x - 1$
- Focus: $x = 2$

• Action:

- Theorems: $x + y = 3 \equiv x = 2$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $y = x - 1$
- Focus: $x = 2$

• Action: zoom out

- Theorems: $x + y = 3 \equiv x = 2$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x + y = 3) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: \emptyset
- Focus: $(x = 2) \wedge (y = x - 1)$
- Action:
- Theorems: $(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = x - 1)$

Window Inference: An Example

- Relation: \equiv
- Context: \emptyset
- Focus: $(x = 2) \wedge (y = x - 1)$
- Action: zoom in on right conjunct
- Theorems: $(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = x - 1)$

Window Inference: An Example

- Relation: \equiv
- Context: $x = 2$
- Focus: $y = x - 1$

• Action:

- Theorems: $y = x - 1 \equiv y = x - 1$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $x = 2$
- Focus: $y = x - 1$

- Action: rewrite using context, and simplify

- Theorems: $y = x - 1 \equiv y = x - 1$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $x = 2$
- Focus: $y = 1$

- Action:

- Theorems: $y = x - 1 \equiv y = 1$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: $x = 2$
- Focus: $y = 1$

- Action: zoom out

- Theorems: $y = x - 1 \equiv y = 1$

$$(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = x - 1)$$

Window Inference: An Example

- Relation: \equiv
- Context: \emptyset
- Focus: $(x = 2) \wedge (y = 1)$
- Action:
- Theorems: $(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = 1)$

Window Inference: An Example

- Relation: \equiv
- Context: \emptyset
- Focus: $(x = 2) \wedge (y = 1)$

- Finished!

- Theorems: $(x + y = 3) \wedge (y = x - 1) \equiv (x = 2) \wedge (y = 1)$

Transformations: Transitivity

- Given relation R , context Γ , focus t , theorem $t_0 R t$
- Transformation step proves $t R t'$
- New theorem is $t_0 R t'$
 - justification: R is transitive
- Initial theorem is $t_0 R t_0$
 - justification: R is reflexive

Zooming: Window Rules

- Paradigm zoom in - transform - zoom out

- Justified by rules of the form
$$\frac{\Delta \vdash s \quad r \quad s'}{\Gamma \vdash t \quad R \quad t'}$$

where s is a subterm of t

- Rule $\wedge L \equiv$ (left conjunct under equivalence):
$$\frac{\Gamma \cup t_2 \vdash t_1 \equiv t'_1}{\Gamma \vdash t_1 \wedge t_2 \equiv t'_1 \wedge t_2}$$

- Read starting up from lower left corner:

- in order to transform $t_1 \wedge t_2$,
- assume t_2 and transform t_1

Structured Derivations

- Window inference gives a dynamic view on proof
- What would a corresponding static view be?
- Our starting point is Calculational proofs
 - a proof format attributed to Feijen and Dijkstra
 - advocated by Gries and Schneider (Logical Approach to Discrete Math)
- Generalised to allow structured proofs and full power of higher-order logic

Linear Proof Steps

- Linear proof steps (calculational proof)

t_1

\sqsubseteq {justification why $t_1 \sqsubseteq t_2$ }

t_2

\sqsubseteq {justification why $t_2 \sqsubseteq t_3$ }

t_3

- Implicit conclusion $t_1 \sqsubseteq t_3$ by transitivity
- Equality steps are also permitted, since $=$ and \sqsubseteq compose to \sqsubseteq

Hierarchical Proof Steps

- Hierarchical proof steps are justified by inference rules

- Rule X has form
$$\frac{\Gamma_1 \vdash t_1 \sqsubseteq t'_1 \dots \Gamma_n \vdash t_n \sqsubseteq t'_n}{\Gamma \vdash t \sqsubseteq t'}$$

- Derivation is written by “turning the rule 90°”

$$\begin{array}{l}
 \Gamma \\
 \vdash t \\
 \sqsubseteq \{\text{rule X}\} \\
 \quad \bullet \Gamma_1 \\
 \quad \vdash t_1 \\
 \quad \sqsubseteq \{\text{justification why } t_1 \sqsubseteq t'_1\} \\
 \quad t'_1 \\
 \quad \vdots \\
 \quad \bullet \Gamma_2 \\
 \quad \vdash t_n \\
 \quad \sqsubseteq \{\text{justification why } t_n \sqsubseteq t'_n\} \\
 \quad t'_n \\
 \dots t'
 \end{array}$$

Generality

- Structured derivations permit
 - general relations
 - subderivations starting on nonsubterms
 - multiple subderivations
- Window inference is much more restricted
- Example: induction by structured derivation

$$\frac{\Gamma \vdash P.0 \Leftarrow \top \quad \Gamma, P.n \vdash P.(n+1) \Leftarrow \top}{\Gamma \vdash (\forall n \bullet P.n) \Leftarrow \top}$$

Structured Derivation: Example

$$x + y = 3 \wedge y = x - 1$$

\equiv {simplify left conjunct}

- $y = x - 1$

$$\vdash x + y = 3$$

\equiv {rewrite using assumption}

$$x + (x - 1) = 3$$

\equiv {simplify}

$$x = 2$$

... $x = 2 \wedge y = x - 1$

\equiv {simplify right conjunct}

- $x = 2$

$$\vdash y = x - 1$$

\equiv {use assumption, and simplify}

$$y = 1$$

... $x = 2 \wedge y = 1$

Structured Derivation: Syntax

- Start of subderivation marked by • (or number)
- Return to outer level marked by ...
- Assumptions separated from initial focus by \vdash
- Justification can be
 - formal: a rule (name, formula, instantiations)
 - informal: an explanation, reference to external information, ...

Browsable Derivations

- A complete derivation shows all details
- Subderivations can be hidden
 - confident readers: hide detailed levels
 - proof sketch: show only top level
- Using html, javascript, etc we can
 - published derivations on the web
 - browse derivations
 - correct and refine derivations

Browsable Derivation: Example

$$x + y = 3 \wedge y = x - 1$$

\equiv {simplify left conjunct}

- $y = x - 1$

$$\vdash x + y = 3$$

\equiv {rewrite using assumption}

$$x + (x - 1) = 3$$

\equiv {simplify}

$$x = 2$$

... $x = 2 \wedge y = x - 1$

\equiv {simplify right conjunct}

- $x = 2$

$$\vdash y = x - 1$$

\equiv {use assumption, and simplify}

$$y = 1$$

... $x = 2 \wedge y = 1$

Browsable Derivation: Example

$$x + y = 3 \wedge y = x - 1$$

\equiv {simplify left conjunct}

$$\dots x = 2 \wedge y = x - 1$$

\equiv {simplify right conjunct}

- $x = 2$

┆ $y = x - 1$

\equiv {use assumption, and simplify}

$$y = 1$$

$$\dots x = 2 \wedge y = 1$$

Browsable Derivation: Example

$$\begin{aligned} & x + y = 3 \wedge y = x - 1 \\ \equiv & \{\text{simplify left conjunct}\} \\ \dots & x = 2 \wedge y = x - 1 \\ \equiv & \{\text{simplify right conjunct}\} \\ \dots & x = 2 \wedge y = 1 \end{aligned}$$

Browsable Derivation: Example

$$x + y = 3 \wedge y = x - 1$$

\equiv {simplify left conjunct}

- $y = x - 1$

$$\vdash x + y = 3$$

\equiv {rewrite using assumption}

$$x + (x - 1) = 3$$

\equiv {simplify}

$$x = 2$$

... $x = 2 \wedge y = x - 1$

\equiv {simplify right conjunct}

... $x = 2 \wedge y = 1$

Example: Conditional

- We define a conditional construct in the logic:

$$\text{if } t \text{ then } t_1 \text{ else } t_2 \text{ fi} = (\epsilon x \cdot (t \Rightarrow x = t_1) \wedge (\neg t \Rightarrow x = t_2))$$

- Definition uses choice operator ϵ (and syntactic sugaring)
- Introduction/elimination rules can be derived

$$\vdash \text{if } \top \text{ then } t_1 \text{ else } t_2 \text{ fi} = t_1$$

$$\vdash \text{if } \text{F} \text{ then } t_1 \text{ else } t_2 \text{ fi} = t_2$$

$$\vdash \text{if } t \text{ then } t' \text{ else } t' \text{ fi} = t'$$

Example: Conditional (Cont'd)

- Subderivation rules can be derived within the logic:

$$\frac{\Gamma \cup t \vdash t_1 = t'_1}{\Gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 \text{ fi} = \text{if } t \text{ then } t'_1 \text{ else } t_2 \text{ fi}}$$

$$\frac{\Gamma \cup \neg t \vdash t_2 = t'_2}{\Gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 \text{ fi} = \text{if } t \text{ then } t_1 \text{ else } t'_2 \text{ fi}}$$

- For zooming in on the condition, the default equality rule is used

$$\frac{\Gamma \vdash t = t'}{\Gamma \vdash \text{if } t \text{ then } t_1 \text{ else } t_2 \text{ fi} = \text{if } t' \text{ then } t_1 \text{ else } t_2 \text{ fi}}$$

Example: Conditional (Cont'd)

- Now we can do derivations with conditionals

if $x = 0$ then $y + x$ else y fi

= {transform left subexpression in context}

- $x = 0$

$\vdash y + x$

= {rewrite using context assumption $x = 0$ }

y

...if $x = 0$ then y else y fi

= {elimination rule}

y

Example: Conditional (Cont'd)

- Now we can do derivations with conditionals

if $x = 0$ then $y + x$ else y fi

= {transform left subexpression in context}

...if $x = 0$ then y else y fi

= {elimination rule}

y

Structured Derivations i High-School Maths

- Proof plays very small role in schools, considered difficult
- When logic is taught, it is treated as a separate subject
- Mathematical problem solving is a form of proof
- Uniform format used in some situations, e.g., equation solving
- We propose to use a derivation format more generally

Why Structured Derivations?

- Strategies and justifications must be stated explicitly
- Solutions can be shared, inspected, and discussed
- Solutions can be refined and updated
- Computer support can be developed
- Only a small effort required to learn the logic needed

Equation Solving

- Original equation is initial expression
- Equation is transformed under equivalence
- Derivation is finished when result is “simple enough”
 - $x = e$ indicates solution (if x does not appear in e)
 - F indicates there are no solutions
- Equation pair is (implicit) conjunction

Equation Example

$$(x - 1)(x^2 + 1) = 0$$

$$\equiv \{\text{product rule } ab = 0 \equiv a = 0 \vee b = 0\}$$

$$x = 1 \vee x^2 + 1 = 0$$

$$\equiv \{\text{transform both disjuncts}\}$$

$$x = 1 \vee x^2 = -1$$

$$\equiv \{\text{square is never negative}\}$$

$$x = 1 \vee \mathbf{F}$$

$$\equiv \{\text{rule for falsity } t \vee \mathbf{F} \equiv t\}$$

$$x = 1$$

Comments

- The whole solution process is kept together
- The connection between the two equations is made explicit
- The level of detail can be reduced as students get more proficient
- Logical rules like $t \vee F \equiv t$ must be mastered (no more difficult than $0 + x = x$)
- Multiple (simple) transformations can be done in one step
- The fact that x ranges over the reals is implicit background knowledge

Going Further

- Consider the following typical exercise:

For what values of x is the expression $\frac{x - 1}{x^2 - 1}$ defined?

- We take the problem expression as our starting point and transform it:

" $\frac{x - 1}{x^2 - 1}$ is defined"

- The derivation uses some basic logical notation and rules

Derivation For Definedness

$\frac{x-1}{x^2-1}$ is defined

\equiv {definedness of rational expressions}

$$x^2 - 1 \neq 0$$

\equiv {switch to logical notation}

$$\neg(x^2 - 1 = 0)$$

\equiv {solve equation $x^2 - 1 = 0$ }

$$\dots \neg(x = -1 \vee x = 1)$$

\equiv {de Morgan's law}

$$\neg(x = -1) \wedge \neg(x = 1)$$

\equiv {switch to nonlogical notation}

$$x \neq -1 \wedge x \neq 1$$

Comments

- Derivation takes us from a problem statement to a solution statement
- The original problem was:

For what values of x is the expression $\frac{x-1}{x^2-1}$ defined?

- To solve it, we transformed the problem expression:

$$\text{"}\frac{x-1}{x^2-1}\text{ is defined"} \equiv x \neq -1 \wedge x \neq 1$$

- Conclusion: the expression is defined for all values of x except -1 and 1

Experiments Using Structured Derivations

- Collection of assignments from Finnish Matriculation Exams have been solved
 - cover equations, analysis, algebra, geometry, etc.
 - available in browsable format
- Method used in Kupittaa High School since autumn 2001
 - controlled experiment
 - initial results are encouraging
 - main obstacle is lack of material and computer support

More Advanced Derivation

“the function $f(x) = 2x$ is uniformly continuous”

\equiv {definition of uniform continuity}

$(\forall \epsilon > 0 \cdot \exists \delta > 0 \cdot (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow |f(y) - f(x)| < \epsilon))$

\Leftarrow {transform consequent}

• $\epsilon > 0$

$\vdash (\exists \delta > 0 \cdot (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow |f(y) - f(x)| < \epsilon))$

\Leftarrow {transform innermost consequent}

• $\delta > 0, x - y < \delta, x < y$

$\vdash |f(y) - f(x)| < \epsilon$

\equiv {definition of f }

$|2y - 2x| < \epsilon$

\equiv {simplify using assumption $x < y$ }

$y - x < \epsilon/2$

\Leftarrow {transitivity, assumption $x - y < \delta$ }

$\delta \leq \epsilon/2$

... $(\exists \delta > 0 \cdot (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow \delta \leq \epsilon/2))$

\Leftarrow { \exists -introduction, witness is $\epsilon/2$ }

\vdots

More Advanced Derivation

“the function $f(x) = 2x$ is uniformly continuous”

\equiv {definition of uniform continuity}

$(\forall \epsilon > 0 \cdot \exists \delta > 0 \cdot (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow |f(y) - f(x)| < \epsilon))$

\Leftarrow {transform consequent}

• $\epsilon > 0$

$\vdash (\exists \delta > 0 \cdot (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow |f(y) - f(x)| < \epsilon))$

\Leftarrow {transform innermost consequent}

... $(\exists \delta > 0 \cdot (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow \delta \leq \epsilon/2))$

\Leftarrow { \exists -introduction, witness is $\epsilon/2$ }

$(\epsilon/2 > 0 \wedge (\forall x y \cdot x - y < \delta \wedge x < y \Rightarrow \epsilon/2 \leq \epsilon/2))$

\equiv {simplify using assumptions}

\top

... $(\forall \epsilon > 0 \cdot \top)$

\equiv {simplify}

\top

I. Reasoning in Hierarchies

- A. A framework for hierarchical reasoning
- B. Predicate transformers – a program hierarchy
- C. Reasoning about programs and loop structures

Predicate Transformers – Background

- (Total) correctness $p \{ S \} q$
 - if program S is executed in an initial state where p holds
then execution is guaranteed to terminate in a state where q holds

- Weakest precondition $\text{wp}. S. q$ (introduced by Dijkstra, 1970's)
 - the weakest (largest) predicate p such that $p \{ S \} q$ holds

- Example: the following correctness assertions are valid:

$$x \geq 3 \{ x := x + 2 \} x \geq 3 \quad x = 2 \{ x := x + 2 \} x \geq 3$$

- But $x \geq 1$ is the weakest possible precondition, so $\text{wp}. (x := x + 2). (x \geq 3) = (x \geq 1)$

The Predicate Transformer Hierarchy

- Layers, each with algebra and reasoning tools
- Built by repeated pointwise extension
- Functions $\Sigma \rightarrow \Sigma$: ; id
- Booleans Bool: T F \wedge \vee \neg \Rightarrow
- Predicates $\Sigma \rightarrow \text{Bool}$: true false \cap \cup \neg \subseteq
- Relations $\Sigma \rightarrow \Sigma \rightarrow \text{Bool}$: True False \cap \cup \neg ; Id \subseteq
- Predicate transformers $(\Sigma \rightarrow \text{Bool}) \rightarrow (\Sigma \rightarrow \text{Bool})$:
 magic abort \sqcap \sqcup \neg ; skip \sqsubseteq

States

- State-based programs: state space is a type Σ , state is $\sigma : \Sigma$
- States are accessed and updated via program variables
- Program variable x (of type α) is pair $(\text{val. } x, \text{set. } x)$ with types

$$\text{val. } x : \Sigma \rightarrow \alpha \quad \text{set. } x : \alpha \rightarrow \Sigma \rightarrow \Sigma$$

- $\text{val. } x. \sigma$ returns value of x in σ (abbreviation: $x. \sigma$)
 - $\text{set. } x. a. \sigma$ returns state where value of x is changed to a
- (Recent more general model uses triple $(\text{add. } x, \text{val. } x, \text{del. } x)$)

Reasoning With States

- Notation $\text{var } x, y$ means that x and y satisfy program variable assumptions:
- Essence of program variables

$$\text{val. } x. (\text{set. } x. a. \sigma) = a$$

$$\text{set. } x. a'. (\text{set. } x. a. \sigma) = \text{set. } x. a'. \sigma$$

$$\text{set. } x. (\text{val. } x. \sigma). \sigma = \sigma$$

- Independence between variables

$$\text{val. } y. (\text{set. } x. a. \sigma) = \text{val. } y. \sigma$$

$$\text{set. } x. a. (\text{set. } y. b. \sigma) = \text{set. } y. b. (\text{set. } x. a. \sigma)$$

Expressions

- Expression (of type α) returns value depending on state: $e : \Sigma \rightarrow \alpha$
- Any type α is ok, e is a term in higher-order logic
- Expressions are written using implicit state abstraction
 - under $\text{var } x, y$
 $(x + y + z - 1)$ means $(\lambda\sigma \cdot x.\sigma + y.\sigma + z - 1)$
- Here $+$ and 1 are pointwise extended

State Predicates

- Predicate $p : \Sigma \rightarrow \text{Bool}$ is set of states
- Constants: false true \cap \cup \neg \subseteq
- Defined by pointwise extension

$$\text{false}.\sigma \equiv \text{F} \quad (p \cap q).\sigma \equiv p.\sigma \wedge q.\sigma$$

$$p \subseteq q \equiv (\forall \sigma \bullet p.\sigma \Rightarrow q.\sigma)$$

- Algebra: a complete lattice
- Predicate can be written as boolean expression
 - under $\text{var } x, y, (x + y + z > 0)$ means $(\lambda \sigma \bullet x.\sigma + y.\sigma + z > 0)$

Reasoning With Predicates

- Reasoning about predicates can be reduced to reasoning about boolean expressions
- Syntactic reduction rule

$$\frac{\vdash \bar{p} \Rightarrow \bar{q}}{\vdash p \subseteq q}$$

where \bar{p} is the predicate p as a boolean expression

- Example:

in order to show $(x > y + 1) \subseteq (x > y)$ (program variables!)

we show $x > y + 1 \Rightarrow x > y$ (no program variables!)

State Functions

- State function $f : \Sigma \rightarrow \Sigma$ maps states to states

- Constants: id ;

- Algebra: a monoid

- Assignment notation: $(x := e)$ defined by

$$(x := e).\sigma = \text{set. } x. (e.\sigma).\sigma$$

- Easily extended to multiple assignment $(x_1, \dots, x_n := e_1, \dots, e_n)$

Reasoning With Assignments

- The substitution property is useful for reasoning with assignments

$$\text{var } x, y \vdash e((x := f).\sigma) = e[x := f].\sigma$$

- For example

$$\text{var } x, y \vdash (x + y).(x := x * y).\sigma = (x * y + y).\sigma$$

- Assignment properties can be derived

$$\text{var } x \vdash (x := e); (x := f) = (x := f[x := e])$$

$$\text{var } x, y \vdash (x := e) = (x, y := e, y)$$

Example: Deriving A Rule (1)

$$\begin{aligned} & ((x := e) ; (x := f)). \sigma \\ = & \{\text{definition of function composition}\} \\ & (x := f). ((x := e). \sigma) \\ = & \{\text{definition of assignment}\} \\ & (\text{set. } x. (f. ((x := e). \sigma))). (\text{set. } x. (e. \sigma). \sigma) \\ = & \{\text{program variable property}\} \\ & (\text{set. } x. (f. ((x := e). \sigma))). \sigma \\ = & \{\text{substitution property}\} \\ & (\text{set. } x. (f[x := e]. \sigma)). \sigma \\ = & \{\text{definition of assignment}\} \\ & (x := f[x := e]). \sigma \end{aligned}$$

Example: Deriving A Rule (2)

- The proof was really a structured derivation

- Extensionality rule:
$$\frac{\vdash t.v = t'.v}{\vdash t = t'} \text{ if } v \text{ nfi } t, t'$$

$$(x := e); (x := f)$$

$$= \{\text{extensionality}\}$$

- $((x := e); (x := f)).\sigma$

$$= \{\text{definition of function composition}\}$$

$$(x := f).((x := e).\sigma)$$

$$= \dots \text{etc.} \dots$$

$$(x := f[x := e]).\sigma$$

$$\dots x := f[x := e]$$

State Relations

- State relation $R : \Sigma \rightarrow \Sigma \rightarrow \text{Bool}$ maps states to sets of states
(or, equivalently, pairs of states to booleans)
- Constants: Id ; false true \cap \cup \neg \subseteq
- Algebra: monoid and complete lattice
- Assignment notation: $(x := x' \mid b)$ defined by

$$(x := x' \mid b). \sigma. \sigma' \equiv (\exists x' \bullet \sigma' = \text{set. } x. x'. \sigma \wedge b. \sigma)$$

Reasoning With Relations

- Well-known relation algebra can be used
- Coercing predicates and state functions into relations

$$|p|. \sigma. \sigma' \equiv p. \sigma \wedge \sigma = \sigma'$$

$$|f|. \sigma. \sigma' \equiv f. \sigma = \sigma'$$

$$|(x := e)| = (x := x' \mid x' = e)$$

- Rule example: making a relational assignment more deterministic

$$\frac{\vdash b \Leftarrow b'}{\vdash (x := x' \mid b) \supseteq (x := x' \mid b')}$$

Predicate Transformers

- Predicate transformer $S : (\Sigma \rightarrow \text{Bool}) \rightarrow (\Sigma \rightarrow \text{Bool})$ maps predicates to predicates
- Constants: skip ; abort magic \sqcap \sqcup \neg \sqsubseteq
- Program interpretation
 - S is program statement, according to weakest precondition semantics
 - skip leaves the state unchanged
 - $S ; S'$ is sequence, S executed first and then S'
 - abort establishes no postcondition, not even true
 - magic establishes any postcondition, even false
 - $S \sqcap S'$ is demonic choice, establishes q if both S and S' do
 - $S \sqcup S'$ is angelic choice, establishes q if one of S and S' does

Monotonic Predicate Transformers

- S monotonic means $(\forall p \ q \bullet p \sqsubseteq q \Rightarrow S.p \sqsubseteq S.q)$
- Algebra: monoid ($;$, skip) and complete lattice (abort, magic, \sqcap , \sqcup ; \sqsubseteq)
- Distribution laws

$$\text{abort} ; S = \text{abort}$$

$$\text{magic} ; S = \text{magic}$$

$$(S_1 \sqcap S_2) ; S = S_1 ; S \sqcap S_2 ; S$$

$$(S_1 \sqcup S_2) ; S = S_1 ; S \sqcup S_2 ; S$$

$$S ; (S_1 \sqcap S_2) \sqsubseteq S ; S_1 \sqcap S ; S_2$$

$$S ; (S_1 \sqcup S_2) \sqsupseteq S ; S_1 \sqcup S ; S_2$$

- Monotonicity laws

$$S_1 \sqsubseteq S'_1 \wedge S_2 \sqsubseteq S'_2 \Rightarrow S_1 ; S_2 \sqsubseteq S'_1 ; S'_2 \quad \text{etc}$$

Importance Of Monotonicity

- Monotonicity laws justify refinement subderivations

$S_1 ; S_2$

\sqsubseteq {refine second component}

- S_2

\sqsubseteq {justification...}

S'_2

... $S_1 ; S'_2$

Predicate Transformer Coercions

- Predicate p can be coerced into predicate transformer
 - assertion $\{p\}$ acts as skip if p holds, as abort otherwise
 - guard $[p]$ acts as skip if p holds, as magic otherwise
- State function f can be coerced into predicate transformer $\langle f \rangle$
 - ordinary assignment statement is $\langle x := e \rangle$
- Relation R can be coerced into predicate transformer $\{R\}$ or $[R]$
 - angelic assignment $\{x := x' \mid b\}$ chooses final state angelically
 - demonic assignment $[x := x' \mid b]$ chooses final state demonically

Predicate Transformer Definitions

$$\text{abort}. q = \text{false}$$

$$\text{skip}. q = q$$

$$\text{magic}. q = \text{true}$$

$$\{p\}. q = p \cap q$$

$$[p]. q = \neg p \cup q$$

$$\langle f \rangle. q = (\lambda \sigma \cdot q.(f.\sigma))$$

$$\{R\}. q = (\lambda \sigma \cdot \exists \sigma' \cdot R.\sigma.\sigma' \wedge q.\sigma')$$

$$[R]. q = (\lambda \sigma \cdot \forall \sigma' \cdot R.\sigma.\sigma' \Rightarrow q.\sigma')$$

$$(S ; S'). q = S.(S'.q)$$

$$(S \sqcap S'). q = S.q \cap S'.q$$

$$(S \sqcup S'). q = S.q \cup S'.q$$

$$S \sqsubseteq S' \equiv (\forall q \cdot S.q \subseteq S'.q)$$

Predicate Transformer Homomorphisms

- Predicates

$$\{p\} ; \{q\} = \{p \cap q\} \qquad [\text{false}] = \text{magic} \qquad \text{etc}$$

$$p \subseteq q \Rightarrow \{p\} \sqsubseteq \{q\} \qquad p \supseteq q \Rightarrow [p] \sqsubseteq [q]$$

- Functions

$$\langle f \rangle ; \langle g \rangle = \langle f ; g \rangle \qquad \langle Id \rangle = \text{skip}$$

- Relations

$$\{Q\} ; \{R\} = \{Q ; R\} \qquad [Q] \sqcap [R] = [Q \cup R] \qquad \text{etc}$$

$$Q \subseteq R \Rightarrow \{Q\} \sqsubseteq \{R\} \qquad Q \supseteq R \Rightarrow [Q] \sqsubseteq [R]$$

Recursion

- Predicate transformers are complete lattice,
so any monotonic function on predicate transformers has a least fixpoint (Tarski)

- Least fixpoint operator μ has characteristic properties

$$f.(\mu.f) = \mu.f \qquad f.S \sqsubseteq S \Rightarrow \mu.f \sqsubseteq S$$

- Write $(\mu X \cdot S)$ for $\mu.(\lambda X \cdot S)$
- Intuition: execute S , and when X is reached, execute S , etc
- Algebra: $f \sqsubseteq g \Rightarrow \mu.f \sqsubseteq \mu.g$

I. Reasoning in Hierarchies

- A. A framework for hierarchical reasoning
- B. Predicate transformers – a program hierarchy
- C. Reasoning about programs and loop structures

Reasoning About Programs

- Program refinement $S \sqsubseteq S'$ can be proved
 - with structured derivations
 - by transforming subexpressions
 - using the predicate transformer hierarchy
- We still need some fine-tuning:
 - rules for conditionals and loops
 - rules for context information

Conditionals

- Deterministic conditional:

$$\text{if } g \text{ then } S_1 \text{ else } S_2 \text{ fi} = [g]; S_1 \sqcap [\neg g]; S_2$$

- Dijkstra's nondeterministic conditional:

$$\text{if } g_1 \rightarrow S_1 \parallel g_2 \rightarrow S_2 \text{ fi} = \{g_1 \cup g_2\}; ([g_1]; S_1 \sqcap [\neg g_1]; S_2)$$

- Conditionals are monotonic: if $S_1 \sqsubseteq S'_1$ and $S_2 \sqsubseteq S'_2$ then

$$\text{if } g \text{ then } S_1 \text{ else } S_2 \text{ fi} \sqsubseteq \text{if } g \text{ then } S'_1 \text{ else } S'_2 \text{ fi}$$

- Introduction and elimination as for earlier conditional:

$$\text{if true then } S_1 \text{ else } S_2 \text{ fi} = S_1 \quad \text{etc.}$$

Iterations

- We define two iteration operators S^* (weak) and S^ω (strong)
 - S^* is finite iteration (zero or more times)
 - S^ω is finite or infinite (looping/aborting) iteration
- Defined as fixpoints: $S^* = (\nu X \cdot S ; X \sqcap \text{skip})$ and $S^\omega = (\mu X \cdot S ; X \sqcap \text{skip})$

$$S^* = S ; S^* \sqcap \text{skip} \qquad S \sqsubseteq T ; S \sqcap \text{skip} \Rightarrow S \sqsubseteq T^*$$

$$S^\omega = S ; S^\omega \sqcap \text{skip} \qquad T ; S \sqcap \text{skip} \sqsubseteq S \Rightarrow T^\omega \sqsubseteq S$$

- Weak iteration is a Kleene star
- Strong iteration is the interesting one, for total correctness

Loops

- Loop constructs are defined using strong iteration
- While loop (deterministic)

$$\text{do } g \rightarrow S \text{ od} = ([g]; S)^\omega; [\neg g]$$

- General loop (nondeterministic)

$$\text{do } g_1 \rightarrow S_1 \parallel g_2 \rightarrow S_2 \text{ od} = ([g_1]; S_1 \sqcap [g_2]; S_2)^\omega; [\neg g_1 \cap \neg g_2]$$

- Loops are monotonic

$$S \sqsubseteq S' \Rightarrow \text{do } g \rightarrow S \text{ od} \sqsubseteq \text{do } g \rightarrow S' \text{ od}$$

Assertions And Context

- A replacement $T[S] \sqsubseteq T[S']$ can be valid, even though $S \sqsubseteq S'$ does not hold
- Refinement in context works in two steps
 - Add assertion with state information: $T[S] = T[\{p\}; S]$
 - Refine $\{p\}; S \sqsubseteq S'$ to conclude : $T[\{p\}; S] \sqsubseteq T[S']$
- Example: a rule for demonic assignment

$$\frac{p \vdash b \Leftarrow b'}{\vdash \{p\}; [x := x' \mid b] \sqsubseteq [x := x' \mid b']}$$

- Dual theory can be built using guards instead

Getting Assertions Right

- Assertions can be introduced and eliminated:

$$\text{skip} = \{\text{true}\} \quad \{p\} \sqsubseteq \text{skip}$$

- Separate rules for pushing assertions forward, e.g.

$$\{p\} ; \langle x := e \rangle = \langle x := e \rangle ; \{\exists x_0 \bullet x = e[x := x_0] \wedge p[x := x_0]\}$$

$$\{p\} ; \text{if } g \text{ then } S_1 \text{ else } S_2 \text{ fi} = \text{if } g \text{ then } \{p \cap g\} ; S_1 \text{ else } \{p \cap \neg g\} ; S_2 \text{ fi}$$

$$\text{if } g \text{ then } S_1 ; \{p_1\} \text{ else } S_2 ; \{p_2\} \text{ fi} = \text{if } g \text{ then } S_1 \text{ else } S_2 \text{ fi} ; \{p_1 \cup p_2\}$$

Back To The Initial Example

- Recall the motivating example:

$\text{do } x > 0 \rightarrow \langle y := y + x \rangle ; \{x > 0\} ; [x := x' \mid 0 \leq x' < x] \text{ od}$

- This program
 - contains predicate, function and relation components
 - lifted to predicate transformers by coercion
 - composed using sequence and loop

The Example: top level

do $x > 0 \rightarrow \langle y := y + x \rangle ; [x := x' \mid 0 \leq x' < x]$ od
= {introduce and propagate context assertion}
do $x > 0 \rightarrow \langle y := y + x \rangle ; \{x > 0\} ; [x := x' \mid 0 \leq x' < x]$ od
 \sqsubseteq {refine substatement}
... do $x > 0 \rightarrow \langle y := y + x \rangle ; \langle x := x - 1 \rangle$ od

The Example: more detail

do $x > 0 \rightarrow \langle y := y + x \rangle ; [x := x' \mid 0 \leq x' < x]$ od
= {introduce and propagate context assertion}
do $x > 0 \rightarrow \langle y := y + x \rangle ; \{x > 0\} ; [x := x' \mid 0 \leq x' < x]$ od
 \sqsubseteq {refine substatement}
• $\{x > 0\} ; [x := x' \mid 0 \leq x' < x]$
 \sqsubseteq {transform demonic assignment in context}
... $[x := x' \mid x' = x - 1]$
= {coercion}
 $\langle x := x - 1 \rangle$
... do $x > 0 \rightarrow \langle y := y + x \rangle ; \langle x := x - 1 \rangle$ od

The Example: full derivation

do $x > 0 \rightarrow \langle y := y + x \rangle ; [x := x' \mid 0 \leq x' < x]$ od
= {introduce and propagate context assertion}
do $x > 0 \rightarrow \langle y := y + x \rangle ; \{x > 0\} ; [x := x' \mid 0 \leq x' < x]$ od
 \sqsubseteq {refine substatement}
• $\{x > 0\} ; [x := x' \mid 0 \leq x' < x]$
 \sqsubseteq {transform demonic assignment in context}
• $x > 0$
 $\vdash 0 \leq x' < x$
 \Leftarrow {arithmetic}
 $x' = x - 1$
... $[x := x' \mid x' = x - 1]$
= {coercion}
 $\langle x := x - 1 \rangle$
... do $x > 0 \rightarrow \langle y := y + x \rangle ; \langle x := x - 1 \rangle$ od

Rules Used In The Example

- Assertion introduction and propagation
- Monotonicity of loop and sequential composition
- Refinement in context for demonic assignment

$$\frac{\Gamma, t_0 \vdash t \Leftarrow t'}{\Gamma \vdash \{t_0\}; [x := x' \mid t] \sqsubseteq [x := x' \mid t']}$$

- Basic arithmetic
- Coercions $|(x := e)| = (x := x' \mid x' = e)$ and $[|f|] = \langle f \rangle$

Verifying Loop Transformations

- Now consider verification of transformation rules for loops
- Examples of rules:

$$\text{do } g \cap h \rightarrow S \text{ od} ; \text{do } g \rightarrow S \text{ od} = \text{do } g \rightarrow S \text{ od}$$

$$\text{do } g \rightarrow S \parallel h \rightarrow T \text{ od} = \text{do } g \rightarrow S ; \text{do } h \rightarrow T \text{ od} \text{ od} \quad \text{if } \dots$$

- Strategy: use algebras for ω and for guards

Iteration Algebra

- First, we have an algebra for ω

$$\text{skip}^\omega = \text{abort}$$

$$\text{magic}^\omega = \text{skip}$$

$$S^\omega ; S^\omega = S^\omega$$

$$S ; (T ; S)^\omega = (S ; T)^\omega ; S$$

leapfrog, sliding

$$(S \sqcap T)^\omega = S^\omega ; (T ; S^\omega)^\omega = (S^\omega ; T)^\omega ; S^\omega$$

decomposition, denesting

$$S \sqsubseteq T \Rightarrow S^\omega \sqsubseteq T^\omega$$

Guard Algebra

- Second, we have an algebra of guards

$$[\text{false}] = \text{magic} \qquad [\text{true}] = \text{skip}$$

$$[p] ; [q] = [p \cap q] \qquad [p] \sqcap [q] = [p \cup q]$$

$$p \supseteq q \Rightarrow [p] \sqsubseteq [q]$$

- Further rules are easily derived when needed

$$[p] ; [p] = [p] \qquad [p] ; [\neg p] = \text{magic}$$

$$[p] ; [q] = [q] ; [p] \qquad \text{etc}$$

Loop Algebra

- Combining iteration algebra and guard algebra
- Further rules, e.g., $[g]([\neg g]; S)^\omega = [g]$
- A general strategy for proving properties of loop structures:
 1. Restate problem in terms of strong iteration and guards
 2. Do derivation while collecting conditions
 3. Restate result in terms of loops and correctness

Example: Nested Loop

- Under what conditions is the following true?

$$\text{do } g \rightarrow S \parallel h \rightarrow T \text{ od} = \text{do } g \rightarrow S ; \text{do } h \rightarrow T \text{ od od}$$

- Restate using iterations and guards, with initialisation I :

$$I([g]S \sqcap [h]T)^\omega [\neg g \cap \neg h] = I([g]S([h]T)^\omega [\neg h])^\omega [\neg g]$$

- Leave ; implicit to make expressions easier to read and write
- Now attempt a derivation

Example: Derivation

$$\begin{aligned} & I([g]S \sqcap [h]T)^\omega [\neg g \cap \neg h] \\ = & I([h]T)^\omega ([g]S([h]T)^\omega)^\omega [\neg g \cap \neg h] \\ = & I[\neg h]([h]T)^\omega ([g]S([h]T)^\omega)^\omega [\neg g \cap \neg h] && \text{if } I = I[\neg h] \\ = & I[\neg h]([g]S([h]T)^\omega)^\omega [\neg g \cap \neg h] \\ = & I[\neg h]([g]S([h]T)^\omega)^\omega [\neg h][\neg g] \\ = & I[\neg h][\neg h]([g]S([h]T)^\omega)^\omega [\neg h][\neg g] && \text{if } [g] = [\neg h][g] \\ = & I[\neg h][\neg h]([g]S([h]T)^\omega [\neg h])^\omega [\neg g] \\ = & I[\neg h]([g]S([h]T)^\omega [\neg h])^\omega [\neg g] \\ = & I([g]S([h]T)^\omega [\neg h])^\omega [\neg g] && \text{if } I = I[\neg h] \end{aligned}$$

Example: Conclusion

- Analysis of conditions:

$I = I[\neg h]$ holds iff initialisation I establishes $\neg h$

$$[g] = [\neg h][g] \equiv g = g \cap \neg h \equiv g \cap h = \emptyset$$

- Thus we have shown that

$$\text{do } g \rightarrow S \parallel h \rightarrow T \text{ od} = \text{do } g \rightarrow S ; \text{do } h \rightarrow T \text{ od od}$$

holds if

- h is initially false, and
- g and h are disjoint

Appendix: Axiomatic Loop Algebra

- Our loop algebra can be axiomatised: Demonic Refinement Algebra
- DRA is similar to Kleene algebra with tests (Kozen, Cohen)
- Axiomatisation: $(;, 1, \sqcap, \sqcup, *, \omega)$ is idempotent semiring with two iteration operators
 - Kleene algebra: $(\cdot, 1, +, 0, *)$
 - Tests: a boolean subalgebra (with complement $\bar{\cdot}$)
- Cohen's Omega algebra adds 2nd iteration ω conservatively (relational model)
- We drop one axiom of KA, have conjunctive (demonic) predicate transformers as model

Axioms of DRA

$$x(yz) = (xy)z$$

$$1x = x$$

$$x1 = x$$

$$\top x = \top$$

$$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$$

$$x \sqcap y = y \sqcap x$$

$$x \sqcap x = x$$

$$\top \sqcap x = x$$

$$x(y \sqcap z) = xy \sqcap xz$$

$$(x \sqcap y)z = xz \sqcap yz$$

$$x^* = xx^* \sqcap 1$$

$$x^* = x^*x \sqcap 1$$

$$x^\omega = xx^\omega \sqcap 1$$

$$x^\omega = x^* \sqcap x^\omega \top$$

$$y \sqsubseteq xy \sqcap z \Rightarrow y \sqsubseteq x^*z$$

$$y \sqsubseteq yx \sqcap z \Rightarrow y \sqsubseteq zx^*$$

$$xy \sqcap z \sqsubseteq y \Rightarrow x^\omega z \sqsubseteq y$$

Axioms of KA

$$x(yz) = (xy)z$$

$$1x = x$$

$$x1 = x$$

$$\top x = \top = x\top$$

$$x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z$$

$$x \sqcap y = y \sqcap x$$

$$x \sqcap x = x$$

$$\top \sqcap x = x$$

$$x(y \sqcap z) = xy \sqcap xz$$

$$(x \sqcap y)z = xz \sqcap yz$$

$$x^* = xx^* \sqcap 1$$

$$x^* = x^*x \sqcap 1$$

$$y \sqsubseteq xy \sqcap z \Rightarrow y \sqsubseteq x^*z$$

$$y \sqsubseteq yx \sqcap z \Rightarrow y \sqsubseteq zx^*$$

Example: Redoing The Derivation

$$\begin{aligned} & i(px \sqcap qy)^\omega \bar{p} \bar{q} \\ = & i(qy)^\omega (px(qy)^\omega)^\omega \bar{p} \bar{q} \\ = & i\bar{q}(qy)^\omega (px(qy)^\omega)^\omega \bar{p} \bar{q} \quad \text{if } i = i\bar{q} \\ = & i\bar{q}(px(qy)^\omega)^\omega \bar{p} \bar{q} \\ = & i\bar{q}(px(qy)^\omega)^\omega \bar{q} \bar{p} \\ = & i\bar{q}(\bar{q}px(qy)^\omega)^\omega \bar{q} \bar{p} \quad \text{if } p = \bar{q}p \\ = & i\bar{q}\bar{q}(px(qy)^\omega \bar{q})^\omega \bar{p} \\ = & i\bar{q}(px(qy)^\omega \bar{q})^\omega \bar{p} \\ = & i(px(qy)^\omega \bar{q})^\omega \bar{p} \quad \text{if } i = i\bar{q} \end{aligned}$$

Epilogue

- The lectures were based on
 - R.J.R. Back and J. von Wright. *Refinement Calculus – a Systematic Introduction*. Springer 1998.
 - R.J.R. Back and J. von Wright. *Structured derivations – a method for doing high-school mathematics carefully*, TUCS report 246, 1999.
 - R.J.R. Back and J. von Wright. *Reasoning algebraically about loops*. *Acta Informatica* 36(295–334), 1999 (also available as TUCS report 144).